Loquendo™ TTS

Multilanguage Text-to-speech Synthesizer

6.5

SDK Programmer's Guide

LoquendoTTS

6.5

SDK Programmer's Guide

Version 6.5.5

21 February 2006

Contents

1 Introduction

1.1 Contents

The present guide is designed for programmers who intend to develop applications using the Loquendo™ TTS - Text-To-Speech synthesizer. This manual is organized in 16 chapters and 2 appendixes:

1.	**CHAPTER 1: Introduction** (this chapter, a preliminary description of the Loquendo TTS SDK including set-up procedure, hardware and software requirements and two minimal sample applications)
2.	**CHAPTER 2: The Loquendo TTS package** (a description of the Loquendo TTS SDK contents)
3.	**CHAPTERS 3-10: The tts APIs** (a detailed description of the Loquendo TTS legacy APIs)
4.	**CHAPTER 11: Mixed Language Configuration**
5.	**CHAPTER 12: Application callback and Loquendo TTS Events** (Description of the events fired by Loquendo TTS)
6.	**CHAPTER 13: Session and Instance Configuration parameters** (How to configure Loquendo TTS)
7.	**CHAPTER 14: Migration from Actor 5.x** (how to easily port an application based on Actor 5.x to Actor or Loquendo TTS 6.x)
8.	**CHAPTER 15: Microsoft SAPI 5 support** (Some notes that may help working with Loquendo SAPI 4 interfaces)
9.	**CHAPTER 16: Microsoft SAPI 4 support** (Some notes that may help working with Loquendo SAPI 5 interfaces)
10.	**CHAPTER 17: Audio destination** (How to manage the speech output, including how to develop a custom audio destination)
11.	**CHAPTER 18: Loquendo TTS ActiveX** (a reference guide for the Loquendo TTS ActiveX describing methods, properties and events)
12.	**CHAPTER 19: Loquendo TTS protection schema** (real-time or batch mode?)
13.	**APPENDIX A: Software Redistribution** (The Loquendo TTS files to be shipped with the final application)
14.	**APPENDIX B: FAQ and Troubleshooting** (A list of the most common Loquendo TTS problems and their possible solutions)

Please refer to the "Loquendo TTS User's Guide" for any information about the following items:

- Loquendo TTS configuration
- Text, sentences and Control Tags
- Use of lexicons and phonetic advanced features

1.2 What is Loquendo TTS?

Loquendo TTS is a Multilanguage/Multi-voice Text-To-Speech synthesizer, peculiar for its very high audio quality and its linguistic accuracy. The Text-to-speech conversion is a real-time “software-only” process: the number of channels that may be served simultaneously depends on the voice quality and the CPU power.

Loquendo TTS is shipped in the form of a set of libraries, both for Windows and Linux, and all its features are accessed by a collection of legacy APIs, that allow controlling of every aspect of the TTS process. The speech can be output to a multimedia audio board, a telephone card or a file. In order to use “custom audio destinations” (such as a LAN, or a legacy audio board) the audio destination developer or vendor can provide its own set of callback functions to be interfaced with the Loquendo TTS library (see chapter 17.1 for details on how to develop a custom audio destination).

Loquendo TTS engine is also compliant to Microsoft Speech SDK 4.0 and Microsoft Speech SDK 5.1 (SAPI). All the “required” interfaces are supported, as well as some “optional” ones. This means that any application using the SAPI TTS interfaces is virtually compatible with Loquendo TTS (see Chapters 15 and 16 for the list of SAPI interfaces supported by the present Loquendo TTS release).

1.3 Hardware and Software Requirements

Loquendo TTS is shipped in the form of a CD (labeled “Loquendo TTS SDK”) containing two different sets of DLLs for Windows (implementing the **tts** APIs, and the Microsoft SAPI 4.0 and 5.1), and a set of Linux shared objects. An additional CD (labeled “Mixed Language Support for Loquendo TTS”) may be present: this software has been first introduced with Loquendo TTS v.6. Additional DLLs for the most popular audio destinations are included in the SDK CD. Different voices are available, with sensibly different qualities – from “nearly robotic” to “nearly human”; basically higher quality voices require more RAM/disk space, but in some way, CPU time can be affected too. Each Loquendo TTS voice is available in 5 different formats with different RAM/disk space requirements:

Audio format	Disk space
16 KHz Linear PCM (tape quality for Multimedia applications)	Up to 250 Mb
11025 Hz Linear PCM (radio quality for Multimedia applications)	Up to 160 Mb
8 KHz Linear PCM (telephone quality)	Up to 120 Mb
8 KHz PCM A-law (telephone quality)	Up to 60 Mb
8 KHz PCM µ-law (telephone quality)	Up to 60 Mb

The suggested hardware/software requirements are:

- CPU Intel Pentium 1000 MHz or more[1]
- 512 Mb RAM[2]
- About 200/250 Mb disk space for each voice
- Windows NT 4.x, 2000, XP or Server 2003 – Linux

[1] The CPU power can limit the number of simultaneous TTS instances. The more powerful it is, the more number of instances can be performed.

[2] The amount of required RAM or disk space is strictly dependent on the speech database sampling frequency and coding. Different voices may also require sensibly different amounts of RAM and disk space. Therefore the specified values are just indicative.

Please refer to the shipped documentation (if any), for Windows or Linux-specific information. Unless differently specified, however, this manual (specifically the API documentation) applies to all available Loquendo TTS versions.

Since the entire system is written in ANSI-C, the Loquendo TTS library may be virtually portable to any architecture supporting this language, including DSP boards[3].

1.4 Set-up for Windows and Unix/Linux

The Loquendo TTS SDK is shipped as one or more CdRoms. The first one (Loquendo TTS SDK) contains the Loquendo TTS DLLs, the manuals and some sample applications: two Italian "robotic" voices are included. **This CD must be installed first[4]**. To install it, see the information inside the "Loquendo TTS SDK Installation Manual".

You may want to install additional high quality voices. Every additional voice will require installing an additional CD; the "Voice" CDs are labeled with a person name (e.g. Susan). The speech databases may be huge; however you don't need to install all databases shipped with each voice (you can select between 5 different sampling frequency and coding – see "Hardware and Software Requirements").

1.5 Versioning

Don't mix different Loquendo TTS releases!

- The Loquendo TTS version number is a three digit string (x.y.z)
- The SDK CD label should report this number
- The version number should appear as soon as you run the installation procedure

1.5.1 What do those digits mean?

The first digit is the "major version" number. This digit changes infrequently. Two Loquendo TTS releases whose versions differ in the first digit may be completely incompatible. Upgrading to a new major version may require application re-design.

The second digit is the "minor version" number. This number changes whenever Loquendo introduces new important functionalities, or adds new TTS languages. Upgrading to minor version doesn't require application re-design. Differently from previous versions, with version 6 or more, Loquendo TTS SDK and voices whose versions differ in the second digit should be compatible (unless differently specified), so upgrading to a new minor release do not require upgrading of every Loquendo TTS voice.

The third digit is the "distribution" number. This number changes frequently (bug fixing, minor functionality changes, etc.). Loquendo TTS SDK and voices whose versions differ in the third digit are completely compatible. For this reason, voice CDs usually report just two-digits in their version number (e.g. 5.7 or 5.7.x).

1.6 tts APIs vs. SAPI

Three different sets of APIs are available. The first set (formerly known as tts APIs) is a collection of legacy APIs that allow accessing the whole set of Loquendo TTS features, from the basic to the advanced. The tts APIs have been designed for simplifying the integration into multimedia applications as well as complex telephony services. These APIs are available on Windows and Linux in two different flavors: as a set of standard C/C++ APIs and as a collection of C++ classes.

[3] The decision and the opportunity of making a new platforms porting (other than Windows and Linux) are entirely left to Loquendo.

[4] The set-up procedure may require Administrator rights (Windows NT and 2000 only): this will be explained later.

The second set of APIs is designed to be compliant to Microsoft SAPI 5.1 (Speech API). SAPI is a popular standard whose purpose is to speed up multilingual speech application development. A SAPI compliant application is virtually compatible with any SAPI compatible text-to-speech engine.

The third set of APIs is compliant to the old Microsoft SAPI 4.0 (Speech API).

While the tts layer has no real dependency from the O.S. (actually Loquendo TTS for Linux exports the same APIs than the Windows version), SAPI is intrinsically connected with the Microsoft Windows O.S. family (specifically Windows NT 4.x or more, Windows 2000, XP and Server 2003) since the core of SAPI technology is entirely based on COM (Component Object Model).

If you are planning to develop a stand-alone application, which will not make use of a wide set of different text-to-speech engines from different vendors, or if you want to assure the compatibility with different platforms (other than Windows), probably the tts APIs are the best for you.

Instead, if you are developing a Windows application and your goal is to integrate a large number of text-to-speech engines, you should consider using the SAPI layer.

1.7 The simplest C language application with the tts APIs

Here is a very short C sample application: this program is a sort of minimal console application, which demonstrates the very basic functionality of Loquendo TTS APIs. This application reads a single sentence and exits.

```
/*****************************************************/
/*                                                   */
/*          Minimal C sample for Loquendo TTS        */
/*              (with Audio Board output)            */
/*                                                   */
/*****************************************************/

#include <stdio.h>
#include "loqtts.h"     /* Loquendo TTS include file  */

int main(int argc, char *argv[])
{ ttsHandleType hInstance;    /* Instance handle */
  ttsHandleType hVoice;       /* Voice handle */
  ttsResultType err;          /* Error code returned by TTS APIs */

  /* Initializes the LoquendoTTS Instance */
  err = ttsNewInstance(&hInstance, NULL, NULL);
  if (err != tts_OK)
  { fprintf(stderr, "%s\n", ttsGetError(NULL));
    return err;
  }

  /* Sets the voice parameters (Mario is the Italian robotic male
voice) */
  err = ttsNewVoice(&hVoice, hInstance, "Mario", 16000, "l");
  if (err != tts_OK)
  { fprintf(stderr, "%s\n", ttsGetError(hInstance));
    return err;
  }

  /* Sets the audio board destination */
  err = ttsSetAudio(hInstance, "LoqAudioBoard", NULL, "l", 0);
  if (err != tts_OK)
  { fprintf(stderr, "%s\n", ttsGetError(hInstance));
    return err;
  }

  /* Converts text to speech */
  err = ttsRead(
      hInstance,  /* Instance handle */
      "Il sistema di sintesi e` correttamente installato.", /* Input
*/
      TTSBUFFER,  /* "Input" is a text buffer   */
      TTSANSI,    /* Input text is in ANSI      */
      TTSDEFAULT, /* Default ReadingMode        */
      TTSBLOCKING);     /* ttsRead keeps control until the end */
  if (err != tts_OK)
  { fprintf(stderr, "%s\n", ttsGetError(hInstance));
    return err;
  }

  /* Closes the Loquendo TTS instance; the voice will be
     automatically closed */
  (void)ttsDeleteInstance(hInstance);

  return 0;
}
```

Every API used in this sample will be fully described in the appropriate section. However, here is a preliminary description:

API	Description
ttsNewInstance	Opens a Loquendo TTS instance. If successful, creates a valid Loquendo TTS instance handle (hInstance). This handle will be used in every subsequent calls to Loquendo TTS APIs
ttsSetAudio	Choose the multimedia audio destination as output for the synthesized PCM. Here "LoqAudioBoard" is the name of a DLL or Shared Object implementing a valid Loquendo TTS Audio Destination. Requires also the speech coding - Linear ("l"), Alaw ("a") or Mulaw ("u")
ttsNewVoice	Sets the voice parameters on the current Loquendo TTS instance: Speaker name: Mario Sampling frequency: 16,000 Hz Sample coding: PCM linear ("l")
ttsRead	Starts reading of a chunk of text.This API can be widely configured. In the example below is blocking and synchronous: it takes the control for the whole speaking time.
ttsDeleteInstance	Closes the current Loquendo TTS instance, and, as a consequence, any open voice
ttsGetError	Returns an error message string – requires an instance or a voice handle, depending on the scenario in which the error occurred

As you can argue from this sample, in order to initialize and use a Loquendo TTS instance, an Audio destination must be specified (by means of the ttsSetAudio API). Possible audio destinations are:

- LoqAudioBoard (output to a Windows or Linux audio board – requires LoqAudioBoard.dll or LoqAudioBoard.so[5])
- LoqAudioFile (output to a RAW or WAV file – requires LoqAudioFile.dll or LoqAudioBoard.so[6])

After ttsSetAudio, the application doesn't need to cope with the Audio destination anymore. See chapter 17 for a more detailed description of the interaction between an application, Loquendo TTS and its Audio destination.

The function ttsRead, which performs the text-to-speech conversion, may be asynchronous or not. This means that, after its invocation, control may return immediately to the caller (asynchronous mode, or non-blocking) or wait until the end of speech (synchronous mode or blocking). The former behavior relies on the multithreading model used by Loquendo TTS. Many synchronization techniques are available for taking advantage of the multithreading model, (i.e. call-back functions) in order to use Loquendo TTS in event-driven programming in an effective way. This will be discussed in the next section.

1.8 The simplest C++ language application with tts classes

The application above could also be approached using the object-oriented paradigm by using Loquendo TTS classes:

```
/******************************************************/
/*                                                    */
/*          Minimal C++ sample for Loquendo TTS       */
/*              (with classes – OOP)                  */
/*                                                    */
/******************************************************/

#include "loqtts.h"
#include <iostream>


int main()
{
    try {
      // Loads voice Susan and speak a sentence
      CttsInstance i;
      i.SetAudio("LoqAudioBoard", NULL, "l", 0);
      CttsVoice v(&i, "Mario", 16000, "l");
      i.Read("Il sistema di sintesi e` correttamente installato.",
          TTSBUFFER,    // "Input" is a text buffer
          TTSANSI,      // Input text is in ANSI
          TTSAUTODETECT,// Default ReadingMode
          TTSBLOCKING); // keeps control till the end
    }
    catch(CttsError e)
    {
       cout << "Error no. " << e.m_ErrorCode <<
            "\tMessage: " << e.m_ErrorString << "\n";
    }
    return 0;
}
```

Here three different classes are involved:

[5] Windows and Linux respectively

[6] Windows and Linux respectively

- **CttsInstance** (whose methods are equivalent to APIs requiring an instance handle as first argument).
- **CttsVoice** (whose methods are equivalent to APIs requiring a voice handle as first argument)
- **CttsError** (for the exception handling)

See the following table to understand the equivalence between classes/methods and APIs:

Class/method	C/C++ API
CttsInstance::CttsInstance	ttsNewInstance
CttsInstance::~CttsInstance	ttsDeleteInstance
CttsInstance::SetAudio	ttsSetAudio
CttsInstance::Read	ttsRead
CttsVoice::CttsVoice	ttsNewVoice
CttsVoice::~CttsVoice	ttsDeleteVoice
CttsError::CttsError	ttsGetError

In addition, for more complex applications, two other classes are available: CttsSession and CttsLexicon. See the API reference for details.

1.9 Synchronous vs. asynchronous programming model

Before continuing with this introduction, one of the crucial points to discuss is the Loquendo TTS programming model. Usually text-to-speech conversion does not make any sense as a standalone process: instead, a complex program (such as a database, a telephony or multimedia application) may use text-to-speech to output information in a flexible and alternative way (instead of printing text, displaying messages to the screen, or playing recorded audio messages). Since text-to-speech conversion is a time consuming process, a TTS engine must provide a user interface that effectively allows a quick interaction with the application, without blocking the normal program flow.

A TTS conversion can be considered made of two different processes: the speech production (i.e. from text to waveform) and the audio rendering (i.e. playing the produced waveform). The two processes can be partially paralleled, at least if the first is faster than the second. But, even supposing that the first process were instantaneous, the second one for sure is a real-time process (reading a 10 seconds message requires exactly 10 seconds). For instance, suppose that an application has to read a 10 minutes long message. If TTS were a blocking process, the program would be unable to perform other tasks, until the end of the TTS conversion. This would be clearly unacceptable.

To solve this, two different approaches may be used.

1.9.1 Synchronous programming model

In a traditional mono-task environment, there is no other possibility than using a cooperative model. This means to split any long operation into smaller ones, each of them performing a "slice" of the whole task. See this example:

`VeryLongTask` is a function that requires 20 seconds to be executed. As soon as the calling procedure invokes `VeryLongTask`, the program cannot accept user input until the procedure exits. Now, VeryLongTask can be replaced by another function - VeryLongTaskSlice, which is functionally equivalent to the former, assuming that it is called in a loop like this:

```
while(!EndOfTask)
{
      EndOfTask = VeryLongTaskSlice();
      /* here the calling may do something else */
      .....
}
```

The end of the task is controlled by a Boolean variable (EndOfTask), which is the return value of VeryLongTaskSlice. This makes the calling program able to perform other tasks, assuming that each call to VeryLongTaskSlice does not take the control for more than a reasonable amount of time.

This approach can be considered "synchronous" because it uses a standard procedural programming model, it does not rely on multi-process/multithread environments, or specific O.S. dependencies.

In an event driven programming environment, and specifically when multithread is available (for instance under Windows) a completely different approach can be used. This will be explained in the next section.

1.9.2 Asynchronous programming model

Suppose to define a new function VeryLongTask2 in this way:

```
int VeryLongTask2()
{
      _beginthread(VeryLongTask(),....);
}
```

We don't want to enter in details on how to create a new thread (this feature is O.S. dependent), but we assume that the O.S. provides a way to do it (_beginthread is one of the possible choices under Windows). After the _beginthread call, VeryLongTask immediately starts, but the control returns immediately to the caller.

From the caller point of view, VeryLongTask2 is a non-blocking function, thus the application can go on with its job. However the application needs probably to know whether VeryLongTask is still running or not. There is a wide choice of programming techniques that may solve this kind of problems. Callback functions are among the most popular. The caller writes a function that VeryLongTask must invoke to signal the application for specific events (such as "I've finished"). A sort of "callback function setup" must be provided to let VeryLongTask know which user function has to be called. For instance:

```
SetupUserCallback(ThisIsMyCallbackForEnd);
VeryLongTask2();

........


ThisIsMyCallbackForEnd()
{
      /* perform user tasks relative to the end of VeryLongTask */
}
```

In this example, the caller declares the name of his callback function, by using function SetupUserCallback, and passing the function pointer ThisIsMyCallbackForEnd as parameter.

At the end of `VeryLongTask`, a call will be made to an unnamed function like:

```
(*pUserCallback)();
```

The purpose of function `SetupUserCallback` is to assign the `pUserCallback` pointer to `ThisIsMyCallbackForEnd`. So the function `ThisIsMyCallbackForEnd` will be called exactly at the end of `VeryLongTask`.

This example is extremely crude and it is far from covering all the major aspects of the problem; however, one of them must be for sure emphasized: the function `ThisIsMyCallbackForEnd` is called from within a thread context other than the application one. This means that the callback function and part of the application can be executed concurrently. This is not a problem indeed; anyway the caller must be aware of it, to avoid all typical problems usually related to concurrent programming.

1.9.3 Synchronous and asynchronous Loquendo TTS APIs

First, you need to choose what model you want to use. The crucial point is the appropriate use of the ttsRead parameters.

Let us enter in more details. In section 1.7 we have seen a simple application made with a synchronous call. If we substitute the following instruction:

```
    // speak a short sentence
    (void)ttsRead (hInstance,"Sintesi dell'italiano.",
            TTSBUFFER,TTSDEFAULT,TTSDEFAULT,TTSBLOCKING);
```

with a loop like this:

```
//Passes text to be synthesized (note that the last parameter has
changed)
(void) ttsRead (hInstance,"Sintesi dell'italiano.",
      TTSBUFFER,TTSDEFAULT,TTSDEFAULT,TTSSLICE);
while(!ttsDone(hInstance))
{
      (void)ttsRead(hInstance,NULL,TTSBUFFER,TTSDEFAULT,
      TTSDEFAULT,TTSSLICE);
      /* other actions */
}
```

we obtain the same effect with a different approach. Here the first call to `ttsRead` sets up the text Loquendo TTS must read, but actually does not perform any TTS conversion. The second call (with NULL as second parameter) performs a “portion” of the whole TTS conversion, until `ttsDone` returns TRUE (ttsDone retrieves the TTS status: a TRUE return value means that TTS has finished).

There is a third possibility: you can pass TTSNONBLOCKING to ttsRead in this case the control returns immediately to the caller. If you try this in the Hello Loquendo TTS sample, you won't hear anything, because the instance will be immediately closed and the program will exit. However, the non-blocking mode should be preferred for developing complex GUI and client/server applications as well as in every event-driven program.

1.10 Sessions and instances

Loquendo TTS 6 introduces the new concept of session/instance/voice architecture.

You can consider a Loquendo TTS session as a common space, in terms of memory and configuration, in which many voices and instances live. Normally an application needs to work with many instances (or channels) of TTS, each of them speaking with different voices and languages. As long as some of those instances use the same voice, or speak the same language, a lot of their data (at least the read-only structures of a single-process application) is shared in order to save memory space.

This is what happens among instances created within the same session. Note that this approach can be used in single or multi-thread applications (e.g. using a different thread for each instance, but just one “shared” session), whilst a session cannot be “shared” between different processes. A Loquendo TTS session is implicitly created as soon as the application creates its first instance.

An application usually doesn’t need to cope with sessions, unless it need to access more than one Loquendo TTS installation (e.g. different Loquendo TTS versions on the same machine), or use different session configurations (see below). Note that in this case there is no memory sharing.

An appropriate API (ttsNewSession) as well as an appropriate class (CttsSession) are provided just in case the application has to manage more than one[7].

A session can be configured by a list of “parameters” that cannot be changed for the whole session life. In order to configure a session, you must specify its “Initialization file” (or is Windows registry section), in which session parameters are stored. The “default.session” file[8] or registry section[9], created during the Loquendo TTS installation, contains all “default” session parameters and usually applications may want to use this one. However you can manage your own session configurations using the appropriate APIs.

Even instances can be configured by parameters listed in an appropriate “IniFile”, but since instances can be configured on the fly, there is no need of a “default.instance” configuration.

1.11 The Simplest SAPI 5 Application

Writing an application with the same functionality as Hello Loquendo TTS is a little more complex with SAPI interfaces. Please see the Microsoft SAPI 5 SDK documentation for a description of the SAPI interfaces. This application will initialize Loquendo TTS with “Mario” voice, read a single sentence and exit (within 5 seconds).

```
/*****************************************************************/
/*                                                               */
/*      Minimal CPP sample for Loquendo TTS SAPI5 interface      */
/*              Speak a short sentence and exit                  */
/*                                                               */
/*****************************************************************/

#include <windows.h>
#include <sapi.h>
#include <sphelper.h>
#include <stdio.h>
#include <atlbase.h>
CComModule _Module;

#include <atlcom.h>


int main(void){
    HRESULT hr;
    ISpVoice * pVoice = NULL;
    IEnumSpObjectTokens *cpEnumVoci=NULL;
    ISpObjectToken *cpVoiceToken=NULL;
    if (FAILED(::CoInitialize(NULL))){
        return -1;
```

[7] This is a Loquendo TTS 6.2 new feature: with previous versions, “explicitly” opening a session was required.
[8] Linux: a file named “default.session”
[9] Windows: [HKEY_LOCAL_MACHINE\SOFTWARE\loquendo\LTTS\default.session]

```
      fprintf(stderr,"Unable to initialize OLE stuff\n");
    }
    hr = CoCreateInstance(CLSID_SpVoice, NULL, CLSCTX_ALL,
IID_ISpVoice, (void **)&pVoice);
    if (FAILED(hr)){
      fprintf(stderr,"Unable to initialize CLSID_SpVoice\n");
        return -1;
    }
    hr = SpEnumTokens(SPCAT_VOICES, L"Name=Mario", NULL,
&cpEnumVoci);
    if (FAILED(hr)){
      fprintf(stderr,"Unable to enumerate voice's token\n");
        return -1;
    }
    hr = cpEnumVoci->Next( 1, &cpVoiceToken, NULL );
    if (FAILED(hr)){
      fprintf(stderr,"Unable to find desidered token\n");
        return -1;
    }
    hr = pVoice->SetVoice( cpVoiceToken);
    if (FAILED(hr)){
      fprintf(stderr,"Unable to set desidered voice\n");
        return -1;
    }
    hr = pVoice->Speak(L"Loquendo TTS SAPI 5: Sintesi
dell'italiano.", SPF_DEFAULT, NULL);
    if (FAILED(hr)){
      fprintf(stderr,"Unable to read sentence\n");
        return -1;
    }
    pVoice->Release();
    pVoice = NULL;
    cpEnumVoci->Release();
    cpEnumVoci = NULL;
    cpVoiceToken->Release();
    cpVoiceToken = NULL;
    ::CoUninitialize();
    return 0;
}
```

1.12 The Simplest SAPI 4 Application

A simple example of application using SAPI 4 is present in the CD distribution.

2 The Loquendo TTS Package

2.1 SDK contents

The Loquendo TTS CD contains the following files for the Windows development:

LoqTTS6.dll **LoqTTS6.lib** **LoqTTS6_util.dll** **LoqLanguageGuesser6.dll**	**LoqTTS6.dll** is the Windows Dynamic link library that implements the tts APIs (The import library LoqTTS6.lib is included for C/C++ projects development – while LoqTTS_util.dll is an additional library that will be dynamically linked by the application). LoqLanguageGuesser6.dll, installed with the CD "Mixed Language Capabilities" (optional) implements the Language Guesser (and it is used by the automatic language detection) LoqTTS6.dll is the only DLL you need to interface with, if you choose the tts APIs.
LoqEnglish6.dll **LoqFrench6.dll** **LoqItalian6.dll** **LoqSpanish6.dll** **LoqGerman6.dll** **LoqPortuguese6.dll** **LoqCatalan6.dll** **LoqSwedish6.dll** **LoqGreek6.dll** **LoqChinese6.dll** **LoqDutch6.dll**	LoqItalian6.dll is a dynamic link library implementing the Italian TTS (included in the Loquendo TTS for testing purposes). Other DLLs for each installed languages will be added with the Loquendo TTS voices.
LoqSAPI5.DLL	Dynamic link library implementing the Loquendo TTS SAPI 5 interfaces.
LoqSAPI4.DLL **LoqAudioSAPI4.DLL**	Dynamic link libraries implementing the Loquendo TTS SAPI 4 interfaces.
LoqActiveXW.ocx **LoqTTS6.lib** **LoqTTS6.tlb**	The Loquendo TTS ActiveX (The import library LoqTTS6.lib is included for C/C++ projects development, while the type library LoqTTS6.tlb is included for Visual Basic development)
LoqAudioBoard.dll	Dynamic link library implementing the Windows Multimedia Audio destination
LoqAudioFile.dll	Dynamic link library implementing the .WAV and RAW File Audio destination. The C source code is included.
Include (folder)	C/C++ header file
DATA (folder)	This folder contains the vocal databases organized in the form of a directory tree: Data/Language/Voice
SAMPLES (folder)	This folder contains several applications that use Loquendo TTS in its different forms. Source code is included for some of them. See next section for details
Edit2Speech.exe	Edit2Speech application. This is a Windows Dialog Application that reads user input interactively, allowing voice, speed, pitch, and lexicon changes.

Eloqwi.exe	Eloqwi application. This is a Windows system tray application that, when enabled, reads interactively the clipboard contents as soon as they change. Can be used in conjunction with any text editor, word processor or web browser
TTSApp.exe	SAPI 5 test application. This is a Microsoft re-distributable application that allows testing of a SAPI engine. See Microsoft SDK documentation for details
AttsTest.exe	SAPI 4 test application. This is a Microsoft re-distributable application that allows testing of a SAPI engine. See Microsoft SDK documentation for details
TTSDirUpdate.EXE	Utility for recovering a bad TTS installation

And for the Linux development:

LoqTTS6.so	This is the Linux shared object that implements the tts APIs. It is the only shared object you need to interface with.
LoqLanguageGuesser6.so	LoqLanguageGuesser6.so, installed with the CD "Mixed Language Capabilities" (optional) implements the Language Guesser (and it is used by the automatic language detection)
LoqEnglish6.so **LoqFrench6.so** **LoqItalian6.so** **LoqSpanish6.so** **LoqGerman6.so** **LoqPortuguese6.so** **LoqCatalan6.so** **LoqSwedish6.so** **LoqGreek6.so** **LoqChinese6.so** **LoqDutch6.so**	LoqItalian6.so is a shared object implementing the Italian TTS (included in the Loquendo TTS for testing purposes). Other shared objects for each installed languages will be added with the Loquendo TTS voices.
LoqAudioBoard.so	Shared object implementing the Linux Audio destination
LoqAudioFile.so	shared object implementing the .WAV and RAW File Audio destination. The C source code is included.
Include (folder)	This directory contains C/C++ header file
Data (folder)	This directory contains the vocal databases organized in the form of a directory tree: Data/Language/Voice
Doc (folder)	This directory contains documentation.

2.2 Sample Applications

For this section see chapter 5 of LoquendoTTS User Guide.

3 Functions Reference

3.1 C/C++ APIs reference

This chapter describes the Loquendo TTS APIs, with their prototypes and their return value. All non-standard types (such as “ttsResultType” and others) are declared in file “loqtts.h”. *Note: unless differently specified, any function argument listed as “char *” or “const char *”, must be considered NULL-terminated.*

From a semantic point of view, the TTS APIs belong to seven different categories:

Categories of LTTS API	
Session and instance	TTS initialization, voice/language loading, audio setting
Control	Data transfer to/from TTS processes (read, pause, resume, skip, stop)
Status and error handling	TTS process configuration and status information
Configuration and Query	Speech parameters settings and configuration, query for installed voices
Prosody	Prosody modifications
Lexicon	Exception handling (expansion, phonetic transcriptions)
Utilities	Text utilities

3.1.1 Instance, Voice and Session

These APIs perform TTS instances opening and closing, global initialization procedures, system global closing, voice settings

Instance Voice and Session APIs	
ttsNewInstance	Opens a new TTS instance on current session and allocates thread resources
ttsDeleteInstance	Closes a TTS instance
ttsNewVoice	Opens a new voice on current instance
ttsDeleteVoice	Closes a voice
ttsActivateVoice	Switch current instance to a different voice
ttsSetAudio	Opens an audio destination and attaches it to current instance
ttsRegisterCallback	Registers a user callback function for asynchronous events handling
ttsEnableEvent	Enable or Disable one TTS event
ttsNewSession	Opens a new TTS session and allocates shared resources
ttsDeleteSession	Closes a TTS session
ttsSetForeignLanguage	Changes the language without changing voice (requires Mixed Language Support)

3.1.2 Control

These APIs address individual TTS instance and activate TTS processes (such as reading, suspend, resume or interrupt speech)

Control APIs	
ttsRead	Synthesizes a text chunk or a text file
ttsStop	Stops current speech
ttsPause	Suspends the TTS conversion (PAUSE)
ttsResume	Resumes a previously suspended TTS conversion (RESUME)
ttsSkip	Skip forward or backward

3.1.3 Status

These APIs retrieve current TTS channel status and/or errors

Status APIs	
ttsGetError	Returns an error message string as soon as an error occurs
ttsDone	Returns the completion status of a TTS conversion
ttsAudioFreeSpace	Returns FALSE when the audio board can accept no more audio samples
ttsSaveStatus	Save the status of all TTS parameters
ttsRecallStatus	Recall the status of all TTS parameters

3.1.4 Configuration

These APIs manage TTS configuration and parameters (audio formats, reading modes, etc.)

Configuration APIs	
ttsLoadConfigurationParam	Loads a keyword value from ini (or registry [Win32])
ttsSaveConfigurationParam	Save a keyword value to a ini (or registry [Win32])
ttsDeleteConfiguration	Remove a ini file or a registry section
ttsGetInstanceParam	Gets the value of a configuration parameter for current instance
ttsSetInstanceParam	Sets the value of a configuration parameter for current instance
ttsGetSessionParam	Gets the value of a configuration parameter for current session
ttsGetVersionInfo	Returns the Loquendo TTS version string
ttsGetDescription	Returns the description string of a voice
ttsSpeakerLanguage	Returns the language of a voice
ttsGetLanguage	Returns the current active language (code + strings)
ttsTestVoice	Tests if a voice is available
ttsGetActiveVoice	Gets the currently active voice
ttsQuery	Lists available voices and gets all their parameters

3.1.5 Prosody

These APIs modify prosody values (such as speed, pitch, volume)

Prosody APIs

ttsSetPitch	Changes the pitch baseline
ttsGetPitch	Get current pitch baseline
ttsSetSpeed	Changes the speech rate
ttsGetSpeed	Get current speech rate
ttsSetVolume	Changes the volume
ttsGetVolume	Get current volume
ttsSetDefaultAttributes	Assigns the default values to all parameters.
ttsSetSpeedRange	Sets up the speed range.
ttsSetPitchRange	Sets up the pitch range.
ttsSetVolumeRange	Sets up the volume range.
ttsGetSpeedRange	Get the speed range.
ttsGetPitchRange	Get the pitch range.
ttsGetVolumeRange	Get the volume range.

3.1.6 Lexicon

These APIs manage exception lexicons (expansions, phonetic transcriptions)

Lexicon APIs	
ttsNewLexicon	Opens a new lexicon for current voice, retrieving it from file
ttsDeleteLexicon	Closes a lexicon, detaching it from current voice
ttsGetLexiconEntry	Queries a lexicon for a specific value
ttsAddLexiconEntry	Adds an entry to the current lexicon
ttsRemoveLexiconEntry	Removes an entry from the current lexicon
ttsSaveLexicon	Saves current lexicon to a file

3.1.7 Utilities

These APIs export some methods for managing phonemes, XML and text

Utility APIs	
ttsLanguageGuess	Detects the language of a chunk of text (requires Mixed Language Support)
ttsPhoneticUtils	Handles phonetic symbols in many ways
ttsValidateXML	Detects if a chunk of text is well XML-formed
ttsPhoneticTranscription	Returns the phonetic transcription of a chunk of text
ttsCheckPhoneticTranscription	Check the syntax of a phonetic string
ttsClaimLicense	Reserve a license (if available)
ttsUnclaimLicense	Release a reserved license

3.2 C++ Classes/methods reference

Five classes are provided for C++ OOP programming:

3.2.1 Class CttsInstance

CttsInstance constructor wraps the ttsNewInstance API, while the destructor wraps ttsDeleteInstance.

A polimorphic implementation of the class constuctor allows the automatic creation of a new session when necessary.

Here is the method list and the equivalent C/C++ API:

Class CttsInstance	Classic API
CttsInstance	ttsNewInstance
~CttsInstance	ttsDeleteInstance
SetAudio	ttsSetAudio
Read	ttsRead
GetParam	ttsGetInstanceParam
SetParam	ttsSetInstanceParam
AudioFreeSpace	ttsAudioFreeSpace
RegisterCallback	ttsRegisterCallback
EnableEvent	ttsEnableEvent
Pause	ttsPause
Resume	ttsResume
Skip	ttsSkip
Done	ttsDone
SaveStatus	ttsSaveStatus
RecallStatus	ttsRecallStatus
GetLanguage	ttsGetLanguage
GetActiveVoice	ttsGetActiveVoice
Query	ttsQuery
SetPitch	ttsSetPitch
GetPitch	ttsGetPitch
SetSpeed	ttsSetSpeed
GetSpeed	ttsGetSpeed
SetVolume	ttsSetVolume
GetVolume	ttsGetVolume
SetDefaultAttributes	ttsSetDefaultAttributes
SetSpeedRange	ttsSetSpeedRange
SetPitchRange	ttsSetPitchRange
SetVolumeRange	ttsSetVolumeRange
GetSpeedRange	ttsGetSpeedRange
GetPitchRange	ttsGetPitchRange
GetVolumeRange	ttsGetVolumeRange
LanguageGuess	ttsLanguageGuess
ValidateXML	tts ValidateXML
PhoneticTranscription	ttsCheckPhoneticTranscription
CheckPhoneticTranscription	ttsCheckPhoneticTranscription
GetExamples	ttsGetExamples

Inject	ttsInject
SetForeignLanguage	ttsSetForeignLanguage
SetModularStructure	ttsSetModularStructure
SetOutput	ttsSetOutput
ClaimLicense	ttsClaimLicense
UnclaimLicense	ttsUnclaimLicense

3.2.2 Class CttsSession

CttsSession constructor wraps the ttsNewSession API, while the destructor wraps ttsDeleteSession.

Here is the method list and the equivalent C/C++ API:

Class CttsSession	Classic API
CttsSession	ttsNewSession
~ CttsSession	ttsDeleteSession
GetParam	ttsGetSessionParam
TestVoice	ttsTestVoice
SpeakerLanguage	ttsSpeakerLanguage
Query	ttsQuery
GetDescription	ttsGetDescription

3.2.3 Class CttsVoice

CttsVoice constructor wraps the ttsNewVoice API, while the destructor wraps ttsDeleteVoice.

Here is the method list and the equivalent C/C++ API:

Class CttsVoice	Classic API
CttsVoice	ttsNewVoice
~ CttsVoice	ttsDeleteVoice
Activate	ttsActivateVoice

3.2.4 Class CttsLexicon

CttsLexicon constructor wraps the ttsNewLexicon API, while the destructor wraps ttsDeleteLexicon.

Here is the method list and the equivalent C/C++ API:

Class CttsLexicon	Classic API
CttsLexicon	ttsNewLexicon
~ CttsLexicon	ttsDeleteLexicon
AddEntry	ttsAddLexiconEntry
RemoveEntry	ttsRemoveLexiconEntry
Save	ttsSaveLexicon

3.2.5 Class CttsError

CttsError constructor wraps the ttsGetError API and can be used for the TTS exeption handling. Use this method in a try-catch costruct.

Here is the method list and the equivalent C/C++ API:

Class CttsError	Classic API
CttsError	ttsGetError

4 Instance, Voice and Session

ttsNewInstance **CttsInstance::CttsInstance** (class constructor)		Opens a new TTS instance

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsNewInstance(`
	`    ttsHandleType *hInstance,`
	`    ttsHandleType hSession,`
	`    const char *IniFile`
	`);`	

C++ Class method(s):	`CttsInstance(`
	`    const char *IniFile=NULL`
	`);`
	`CttsInstance(`
	`    CttsSession &session,`
	`    const char *IniFile=NULL`
	`);`	

Classic C Arguments:	`ttsHandleType *hInstance` **[OUT]** `ttsHandleType hSession` **[IN]** `const char *IniFile` **[IN]**	new instance handle session handle (can be NULL) Ini file name or registry section [Win32][10] – may be NULL (in this case the default configuration is used)
C++ Class differences	`CttsSession &session` **[IN]**	pointer to a previously created CttsSession class Note: instance handle not used in class implementation. Using the 1st class constuctor a ttsSession is automatically created.
Return value:	The classic API returns TTS_OK (zero) in case of success or a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	A valid session handle as hSession argument may be specified or NULL (default). See the examples below IniFile (if any) may contain specific configuration values for this instance, such as “TaggedText” or “LogFile”. See 13.2 for details	

[10] This parameter can be NULL; in this case the default registry section, created during installation, is used: [HKEY_LOCAL_MACHINE\SOFTWARE\loquendo\LTTS\default.session]

- **Description**

Opens a new TTS instance, belonging to a specific session. This API creates an implicit session, if needed, as far as NULL is specified as its 2nd argument. Every subsequent instance will belong to the same session, unless a different session handle is specified (to create a new session see ttsNewSession).

If successful, ttsNewinstance creates a valid session instance (hInstance) to be used in any subsequent call to APIs requiring an instance handle as argument. In case of failure the application should call ttsGetError passing the hSession value (or NULL) as argument, to obtain the error string description.

- **Example 1 (using the implicit session)**

```
ttsHandleType hInstance;
      ttsResultType r = ttsNewInstance(&hInstance,NULL,NULL);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(NULL));
      return –1;
}
```

- **Example 2 (using a different session)**

```
ttsHandleType hInstance;
ttsHandleType hSession;
ttsResultType r = ttsNewSession(&hSession,NULL);
if(tts_OK != r) {
      fprintf(stderr,
       ”Error opening default session. Check Loquendo TTS
      installation\n”);
      return –1;
}
r = ttsNewInstance(&hInstance,hSession,NULL);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hSession));
      return –1;
}
```

- **Example 3 (using implicit session and C++ class)**

```
try {
      // Creating a new instance
      CttsInstance i;
}
catch(CttsError e) {
      Cout << "Error no. " << e.m_ErrorCode <<
            "\tMessage: " << e.m_ErrorString << "\n";
}
```

ttsNewVoice CttsVoice::CttsVoice (class constructor)		Opens a new voice for current instance
Classic C prototype:	ttsResultType tts_API_DEFINITION ttsNewVoice(ttsHandleType *hVoice, ttsHandleType hInstance, const char *Speaker, unsigned int SampleRate, const char *coding);	
C++ Class method(s):	CttsVoice::CttsVoice(const char *IniFile, const char *Speaker, unsigned int SampleRate, const char *coding);	
Classic C Arguments:	ttsHandleType *hVoice **[OUT]**	voice handle
	ttsHandleType hInstance **[IN]**	instance handle
	const char *Speaker **[IN]**	voice (speaker) name
	unsigned int SampleRate **[IN]**	sample rate in hz
	const char *coding **[IN]**	sample coding { "L" (linear), "A" (A-law), "U" (u-law) }
C++ Class differences	voice handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	Requires a valid instance handle as 2nd argument	

- **Description**

Opens a TTS voice and attaches it to current instance. From this moment on, this voice become the "active" one for current instance. More than one voice can be open on a single instance. To change the active voice among the open voices use ttsActivateVoice.

If successful, ttsNewVoice creates a valid voice handle (hVoice) to be used in any subsequent call to APIs requiring a voice handle as argument. In case of failure the application should call ttsGetError passing the hInstance value as argument, to obtain the error string description.

- **Example 1**

```
r = ttsNewInstance(&hInstance,NULL,NULL);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(NULL));
      return –1;
}
r = ttsNewVoice(&hVoice,hInstance,”Susan”,16000,”L”);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hInstance));
      return –1;
}
```

- **Example 2 (same code in C++)**

```
try {
      // Creating a new instance
      CttsInstance i;
      CttsVoice v(&i, ,”Susan”,16000,”L”);
}
catch(CttsError e) {
      Cout << "Error no. " << e.m_ErrorCode <<
            "\tMessage: " << e.m_ErrorString << "\n";
}
```

ttsActivateVoice **CttsVoice::Activate()**		Switch current instance to a different voice

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsActivateVoice(`
	`    ttsHandleType hVoice`
	`);`	

C++ Class method(s):	`void CttsVoice::Activate();`	
Classic C Arguments:	`ttsHandleType hVoice` **`[IN]`**	voice handle
C++ Class differences	voice handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	Requires a previously opened voice as argument	

- **Description**

Switches current instance to a different voice. From this moment on, this voice become the “active” one for current instance. In case of failure the application should call ttsGetError passing the hInstance or the hVoice value as argument, to obtain the error string description.

- **Example 1**

```
......
r = ttsNewVoice(&hVoice1,hInstance,”Susan”,16000,”L”);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hInstance));
      return –1;
}
r = ttsNewVoice(&hVoice2,hInstance,”Dave”,16000,”L”);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hInstance));
      return –1;
}
/* do something with Dave */
...........
/* now switch to Susan */
r = ttsActivateVoice(hVoice1);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hInstance));
      return –1;
}
```

- **Example 2 (same code in C++)**

```
try {
      // Creating a new instance
      CttsInstance i;
      CttsVoice v1(&i, ,”Susan”,16000,”L”);
      CttsVoice v2(&i, ,”Dave”,16000,”L”);
     // do something with dave
     …
     // now switch to Susan
     v2.Activate();
}
catch(CttsError e) {
      Cout << "Error no. " << e.m_ErrorCode <<
            "\tMessage: " << e.m_ErrorString << "\n";
}
```

ttsSetAudio CttsInstance::SetAudio		Opens an audio destination and attaches it to current instance

Classic C Prototype:	ttsResultType tts_API_DEFINITION ttsSetAudio(
	    ttsHandleType hInstance,
	    const char *AudioDestName,
	    const char *AudioDeviceName,
	    const char *coding,
	    const void *pUser
	);	

C++ Class method:	void CttsInstance::SetAudio(
	    const char *AudioDestName,
	    const char *AudioDeviceName,
	    const char *coding,
	    const void *pUser
	);	

Classic C Arguments:	ttsHandleType hInstance **[IN]**	instance handle
	const char *AudioDestName **[IN]**	Name of a DLL or shared object implementing an audio destination
	const char *AudioDeviceName **[IN]**	Name of a valid device name for the requested Audio destination
	const char *coding **[IN]**	sample coding { “L” (linear), “A” (A-law), “U” (u-law) }
	const void *pUser **[IN]**	optional user data
C++ Class differences	instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	3rd and 5th argument values are specific of the requested audio destination library.	

- **Description**

Sets the audio destination for current instance. See chapter 17 for details on how to use audio destination libraries.

In case of failure the application should call ttsGetError passing the hInstance value as argument, to obtain the error string description.

- **Example 1**

```
ttsResultType r =
ttsSetAudio(hInstance,”LoqAudioFile”,”fileout.pcm”,”l”,0);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hInstance));
      return –1;
}
```

- **Example 2 (same code in C++)**

```
try {
      CttsInstance i;
      i.SetAudio("LoqAudioBoard", NULL, "l", 0);
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsRegisterCallback **CttsInstance::RegisterCallback**		Registers the application callback function

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsRegisterCallback(`
	`    ttsHandleType hInstance,`
	`    ttsCallbackType pfnCallback,`
	`    void *pUser,`
	`    unsigned long Reserved`
	`);`	

C++ Class method:	`void CttsInstance::RegisterCallback(`
	`    ttsCallbackType pfnCallback,`
	`    void *pUser,`
	`    unsigned long Reserved`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]**	instance handle
	`ttsCallbackType pfnCallback` **[IN]**	pointer to a callback function
	`void *pUser` **[IN]**	Application-dependent data pointer
	`unsigned long Reserved` **[IN]**	RFU
C++ Class differences	instance handle not used in class implementation	
Return:	tts_OK, (zero) in case of success. Otherwise returns a 32 bits error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instances	

Notes:	The prototype of a valid callback function is:
	`void TTSCALLBACK myCallback(`
	`  ttsEventType nReason,`
	`  void * lData,`
	`  void * pUser`
	`);`	

- **Description**

Registers the callback function for any asynchronous TTS event. See chapter 12 (Application callback and Loquendo TTS Events) for details.

- **Example**

```
MyStruct s;
ttsRegisterCallback(nInstance,myCallback,&s,0);
……
void TTSCALLBACK myCallback(ttsEventType nReason, void *lData, void
*pUser)
{
      switch (nReason) {
            …
      }
}
```

- **Example 2 (same code in C++)**

```
try {
      CttsInstance i;
      i.RegisterCallback(myCallback, &s, 0);
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsEnableEvent **CttsInstance::EnableEvent**		Enable or disable one TTS event
Classic C Prototype:	ttsResultType tts_API_DEFINITION ttsEnableEvent(ttsHandleType hInstance, ttsEventType evt, ttsBoolType bEnabled);	
C++ Class method:	void CttsInstance::EnableEvent(ttsEventType evt, ttsBoolType bEnabled);	
Arguments:	`ttsHandleType hInstance` **[IN]**	instance handle
	ttsEventType evt **[IN]**	TTS event
	ttsBoolType bEnabled **[IN]**	If FALSE the event is disabled, if TRUE enabled
C++ Class differences	instance handle not used in class implementation	
Return:	tts_OK, (zero) in case of success. Otherwise returns a 32 bits error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instances	
Notes:		

- **Description**

If put to "false", disable one of the asynchronous TTS event. If put to "true", enable the TTS event again. See chapter 12 (Application callback and Loquendo TTS Events) for the list of the available events.

ttsNewSession **CttsSession::CttsSession** (class constructor)		Opens a new TTS session and allocates shared resources

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsNewSession(`
	`    ttsHandleType *hSession,`
	`    const char *IniFile`
	`);`	

C++ Class method:	`CttsSession::CttsSession(`
	`    const char *IniFile = NULL`
	`);`	

Arguments:	`ttsHandleType *hSession` **`[OUT]`**
	`const char *IniFile` **`[IN]`**
		new session handle
		Optional Ini file name or registry section [Win32][11] – may be NULL (in this case the default configuration is used)

C++ Class differences	session handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	IniFile (if any) may contain specific configuration values for this session, such as “DataPath” or “LibraryPath”. See 13.2 for details	

- **Description**

Opens a TTS session. Normally there is no need to call this function explicitly because a session is implicitly created as far as the first application's instance is created. However, the API is useful when the application wants to manage its session(s) explicitly.

Normally just one session is needed, no matter of the number of instances, voices and audio channels you want to drive. All instances belonging at the same session share their read/only memory within a single process space; therefore it's a good idea to open just on session per application, unless you need to access different Loquendo TTS installations, use different configurations or access TTS instances from different processes.

If successful, ttsNewSession creates a valid session handle (hSession) to be used in every subsequent call to APIs requiring a session handle as parameter (e.g. ttsNewInstance).

[11] This parameter can be NULL; in this case the default registry section, created during installation, is used: [HKEY_LOCAL_MACHINE\SOFTWARE\loquendo\LTTS\default.session]

- **Example 1**

```
ttsHandleType hSession;

ttsResultType r = ttsNewSession(&hSession,NULL);
if(tts_OK != r) {

      fprintf(stderr,

       ”Error opening default session. Check Loquendo TTS
      installation\n”);

      return –1;
}
```

- **Example 2 (same code in C++)**

```
try {
      CttsSession s;
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsDeleteSession **CttsSession::~CttsSession** (class destructor)		Closes a TTS session

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsDeleteSession(`
	`    ttsHandleType hSession`
	`);`	

C++ Class method:	`void CttsSession::~CttsSession();`	
Arguments:	`ttsHandleType hSession` **`[IN]`**	session handle (can be NULL)
C++ Class differences	session handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	The hSession value can be NULL. In this case the implicit session is referred.	

- **Description**

Closes a TTS session and all its attached voices/instances. In case of failure the application should call ttsGetError passing the hSession value as argument, to obtain the error string description.

The hSession value can be NULL. In this case the implicit session is referred. The following call:

```
ttsDeleteSession(NULL); // closes everything!
```

will close all instances and voices, and free all session memory.

Using C++ class the same concept is:

```
try {
      CttsSession s;
      // ... do something ...
      delete s;
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsDeleteInstance **CttsInstance::~CttsInstance** (class destructor)		Closes a TTS instance

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsDeleteInstance(`
	`    ttsHandleType hInstance`
	`);`	

C++ Class method:	`void CttsInstance::~CttsInstance();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	By calling ttsDeleteSession all attached instances are closed	

- **Description**

Closes a TTS instance and all its attached voices. There is no need to call explicitly this method if you choose to delete its root session.
In case of failure the application should call ttsGetError passing the hInstance value as argument, to obtain the error string description.

- **Example**

```
/* here calling ttsDeleteInstance would be unnecessary */
/* because of the ttsDeleteSession call */
ttsResultType r = ttsDeleteInstance(hInstance);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hInstance));
return –1;
}
r = ttsDeleteSession(hSession);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hSession));
      return –1;
}
```

- **Example 2 (same code in C++)**

```
try {
      CttsSession s;
      CttsInstance I(&s, “”);
      // … do something …
      delete i;
      delete s;
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsDeleteVoice **CttsVoice::~CttsVoice** (class destructor)		Close an open voice

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsDeleteVoice(`
	`    ttsHandleType hVoice`
	`);`	

C++ Class method:	`void CttsVoice::~CttsVoice();`	
Arguments:	`ttsHandleType hVoice` **`[IN]`**	voice handle
C++ Class differences	Voice handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	By calling ttsDeleteInstance all attached voices are closed	

- **Description**

Closes a TTS voice. Normally there is no need to call explicitly this method, as any open instances are closed when closing their root instance (as it happens to all open instances at their session closure). In case of failure the application should call ttsGetError passing the hInstance or the hVoice value as argument, to obtain the error string description.

- **Example 1**

```
/* here calling ttsDeleteVoice would be unnecessary */
/* because of the ttsDeleteInstance call           */
r = ttsDeleteVoice(hVoice);
if(tts_OK != r) {
      fprintf(stderr,"%s\n",ttsGetError(hVoice));
return –1;
}
r = ttsDeleteInstance(hInstance);
if(tts_OK != r) {
      fprintf(stderr,"%s\n",ttsGetError(hInstance));
return –1;
}
```

- **Example 2 (same code in C++)**

```
try {
      CttsInstance i;
      CttsVoice v;
      // … do something …
      delete v;
      delete i;
}
catch(CttsError e) {
      …
}
```

ttsSetForeignLanguage **CttsInstance:: SetForeignLanguage**		Changes Language

C/C++ Prototype:	`ttsResultType tts_API_DEFINITION ttsSetForeignLanguage(`
	`   ttsHandleType hInstance,`
	`   const char* SecondLanguage`
	`)`	

C++ Class method:	`void CttsInstance::SetForeignLanguage(`
	`   const char* SecondLanguage`
	`)`	

Arguments:	`ttsHandleType hInstance` **`[IN]`**
	`const char* SecondLanguage`
		instance handle
		language string identifier (if NULL the voice’s default language is used)

C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Session and instance	
Notes:	Requires the additional CD “Mixed Language Support”	

- **Description**

Changes the active voice’s language. This is one of the features of the Loquendo Mixed Language Support: any voice can be temporarily switched to a “foreign” language. Note that a voice speaking a foreign language cannot have the same the quality than a native one.

Passing NULL (or an invalid language string) resets the voice to its default language. Valid language strings are: “English”, “French”, “German”, “Italian”, “Spanish”, “Greek”, “Swedish”, “Portuguese”, “Catalan”, “Chinese”

- **Example 1**

```
r = ttsSetForeignLanguage(hInstance,”english”);
if(tts_OK != r) {
      fprintf(stderr,”%s\n”,ttsGetError(hInstance));
      return –1;
}
```

- **Example 2 (same code in C++)**

```
try {
      CttsInstance i;
      // … do something …
      i.SetForeignLanguage(“english”);
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

5 Control functions

ttsRead **CttsInstance::Read**		Synthesizes a text chunk or text file

Classic C Prototype:	`ttsRead(`
	`    ttsHandleType hInstance,`
	`    const void *Input,`
	`    int InputType,`
	`    int TextCoding,`
	`    int ReadingMode,`
	`    int ProcessingMode`
	`);`	

C++ Class method:	`void CttsInstance::Read(`
	`    const void *Input,`
	`    int InputType,`
	`    int TextCoding,`
	`    int ReadingMode,`
	`    int ProcessingMode`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]**	instance handle
	`const void *Input` **[IN]**	string text or filename
	`int InputType` **[IN]**	TTSFILE, TTSBUFFER, TTSDEFAULT
	`int TextCoding` **[IN]**	TTSANSI, TTSISO, TTSUTF8, TTSUNICODE, TTSAUTODETECT or TTSDEFAULT
	`int ReadingMode` **[IN]**	TTSMULTILINE, TTSPARAGRAPH, TTSSSML, TTSXML, TTSAUTODETECT or TTSDEFAULT
	`int ProcessingMode` **[IN]**	TTSBLOCKING, TTSNONBLOCKING, TTSSLICE or TTSDEFAULT
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Control	
Notes:	This API must be called on the same thread context where hInstance has been created	

- **Description**

Performs a TTS conversion. Filename can be a valid URL too (supported on Windows, on Linux by means of the library "libcurl.so" usually included in the Linux distributions, not supported on Solaris).

- **Parameters**

InputType	**TTSBUFFER**	**"Input" is a text buffer**
	TTSFILE	**"Input" is the name of a text file**
	TTSDEFAULT	**The content of "Input" depends on the instance configuration parameter: "InputType". If not defined: TTSBUFFER**
TextCoding	**TTSANSI**	**Input text is in Windows ANSI**
	TTSISO	**Input text is ISO Latin (the code page depends on the language)**
	TTSUNICODE	**Input text is UNICODE (UTF-16)**
	TTSUTF8	**Input text is UTF-8**
	TTSAUTODETECT	**Input text coding has to be detected automatically (requires TTSFILE as InputType)**
	TTSDEFAULT	**Input text coding depends on the instance configuration parameter: "InputTextCoding". If not defined: TTSANSI**
ReadingMode	**TTSMULTILINE**	**Input text has no line breaks**
	TTSPARAGRAPH	**Input text has line breaks**
	TTSSSML or TTSXML	**Input text is well formed SSML (req. Voice XML 1.0 or Voice XML 2.0)**
	TTSAUTODETECT	**Input text format has to be detected automatically**
	TTSDEFAULT	**Input text format depends on the instance configuration parameter: "ReadingMode". If not defined: TTSMULTILINE**
ProcessingMode	**TTSBLOCKING**	**ttsRead keeps control until the end of TTS conversion (synchronous mode)**
	TTSNONBLOCKING	**TTS conversion is non-blocking: ttsRead returns immediately to the caller (asynchronous mode). Any other non-blocking ttsRead are buffered and will start playing at the end of current speech conversion**
	TTSSLICE	**ttsRead must be called in a loop with ttsDone (see example below). Any subsequent call to ttsRead must pass NULL as 2nd argument**
	TTSDEFAULT	**Processing mode depends on the instance configuration parameter: "ProcessingMode". If not defined: TTSNONBLOCKING**

- **Example 1**

```
r = ttsReadText(hInstance,”This is a simple text”, TTSBUFFER,
    TTSDEFAULT,
    TTSDEFAULT, TTSSLICE);
if(tts_OK == r)
while(!ttsDone(hInstance))
      ttsReadText(hInstance,NULL, TTSBUFFER, TTSDEFAULT,
         TTSDEFAULT, TTSSLICE);
```

- **Example 2**

```
r  =  ttsReadText(hInstance,”This  is  a  simple  text”,  TTSBUFFER,
     TTSDEFAULT,
     TTSDEFAULT, TTSBLOCKING);
```

- **Example 3 (same code 1 in C++)**

```
try {
      CttsInstance i;
      // … do something …
      i.Read(“This is a simple text”, TTSBUFFER, TTSDEFAULT,
             TTSDEFAULT, TTSSLICE);
      While (!i.ttsDone()) {
            i.Read(NULL, TTSBUFFER, TTSDEFAULT, TTSDEFAULT,
                   TTSSLICE);
      }
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsStop **CttsInstance::Stop**		stops and abort TTS conversion

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsStop(`
	`    ttsHandleType hInstance`
	`);`	

C++ Class method:	`void CttsInstance::Stop();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Control	
Notes:	This API must be called on the same thread context where hInstance has been created	

- **Description**

Stops and abort all buffered non-blocking TTS conversions on current instance.

ttsPause **CttsInstance::Pause**		stops (pauses)TTS conversion

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsPause(`
	`    ttsHandleType hInstance`
	`);`	

C++ Class method:	`void CttsInstance::Pause();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Control	
Notes:	This API must be called on the same thread context where hInstance has been created	

- **Description**

Pauses an active TTS conversion on current instance. Call ttsResume to resume TTS conversion. Requires a run-time audio destination (see 17.1 for details)

ttsResume **CttsInstance::Resume**		resumes a paused TTS conversion

C/C++ Prototype:	`ttsResultType tts_API_DEFINITION ttsResume(`
	`    ttsHandleType hInstance`
	`);`	

C++ Class method:	`void CttsInstance::Resume();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Control	
Notes:	This API must be called on the same thread context where hInstance has been created	

- **Description**

Resume a paused TTS conversion on current instance. Requires a run-time audio destination (see 17.1 for details)

ttsSkip **ttsInstance::Skip**	Skip forward or backward	

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSkip(`
	`    ttsHandleType hInstance,`
	`    unsigned int Type,`
	`    signed int nItems`
	`);`	

C++ Class method:	`void CttsInstance::Skip(unsigned int Type,`
	`    signed int nItems);`	

Arguments:	`ttsHandleType hInstance` **[IN]** `unsigned int mode` **[IN]** `signed int nItems` **[IN]**	instance handle mode of skip action # of items to skip
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Control	
Notes:	This API must be called on the same thread context where hInstance has been created	

- **Description**

Skips backward or forward while an asynchronous TTS conversion is active. Three modes are possible: TTS_PROMPT, TTS_SENTENCE and TTS_GOTOSENTENCE. TTS_PROMPT mode causes the voice to skip forward the specified number of prompts (a prompt is a text chunk passed as the argument of an asynchronous "ttsRead"). Requires that more than a prompt has been buffered by means of asynchronous ttsRead calls (See the examples below). nItems must be grater than zero.

The second (TTS_SENTENCE) causes the voice to move forward or backward by a given number of sentences, starting from the one you are currently hearing; nItems may be greater than zero (skip forward), or less than zero (skip backward): a value of zero has no effect.

The third (TTS_GOTOSENTENCE) is similar to the TTS_SENTENCE one, except than the sentence number is an absolute greater than zero (or equal) value within the prompt. The first sentence of a prompt is numbered as 'zero'.

Skipping sentences out of current prompt is not possible. Too high 'skip' values simply cause a premature end-of-speech or a restart from beginning.

Example 1

```
ttsRead(hInstance,”first chunk”,…..,TTSNONBLOCKING);
ttsRead(hInstance,”second chunk”,…..,TTSNONBLOCKING);
ttsRead(hInstance,”third chunk”,…..,TTSNONBLOCKING);
ttsSkip(hinstance,TTS_PROMPT,2);
```

In the example above the voice would jump immediately from “first chunk” to “third chunk”.

Example 2

```
ttsRead(hInstance,”first sentence. second sentence”, third sentence.
“fourth and last one”. ……, TTSNONBLOCKING);
```

`ttsSkip(hInstance,TTS_GOTOSENTENCE,2);`

In the example above the voice would jump to “third sentence”.

Example 3 (same as 1 but in C++)

```
try {
      CttsInstance i;
      // … do something …
      i.Read(”first chunk”,…..,TTSNONBLOCKING);
      i.Read(”second chunk”,…..,TTSNONBLOCKING);
      i.Read(”third chunk”,…..,TTSNONBLOCKING);
      i.Skip(TTS_GOTOSENTENCE,2);
}
catch(CttsError e) {
      …
}
```

6 Status functions

ttsDone **CttsInstance::Done**	Returns the completion status of a TTS conversion	

Classic C Prototype:	`ttsBoolType tts_API_DEFINITION ttsDone(`
	`    ttsHandleType hInstance`
	`);`	

C++ Class method:	`ttsBoolType CttsInstance::Done();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation	
Return value:	TRUE if current TTS conversion has finished. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Status	
Notes:	This API must be called on the same thread context where hInstance has been created	

- **Description**

Used in conjunction with ttsRead in TTSSLICE mode, ttsDone checks whether current TTS conversion has finished.

- **Example**

```
while (!ttsDone(nInstance)) {
      r = ttsRead(nInstance, NULL, TTSBUFFER, TTSDEFAULT,
                  TTSDEFAULT, TTSSLICE);
      if ( r != tts_OK )
      {
            PrintError(r); /* process error some way */
      }
    } /* until not finished */
```

- **Example 2 (same code 1 in C++)**

```
try {
      // … do something …
      While (!i.ttsDone()) {
            i.Read(NULL, TTSBUFFER, TTSDEFAULT, TTSDEFAULT,
                   TTSSLICE);
      }
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

- **See also**

ttsRead, ttsAudioFreeSpace

ttsAudioFreeSpace **CttsInstance::AudioFreeSpace**		Check whether the audio destination internal buffers can accept more data

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsAudioFreeSpace(`
	`    ttsHandleType hInstance,`
	`    ttsBoolType *bFreeSpace`
	`);`	

C++ Class method:	`void CttsInstance::AudioFreeSpace (`
	`    ttsBoolType *bFreeSpace`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]** `ttsBoolType *bFreeSpace` **[OUT]**	instance handle TRUE if there is room
C++ Class differences	Instance handle not used in class implementation. Use CttsError class for C++ exception handling.	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code	
Inclusions:	loqtts.h	
Category:	Status	
Notes:	This API must be called on the same thread context where hInstance has been created	

- **Description**

Check if the audio destination internal buffers can accept more data or not (the result is stored in bFreeSpace as a Boolean value). This API is useful to decrease the CPU consumption when very long texts are sent to an audio board.

- **Example 1**

```
while (!ttsDone(nInstance)) {
      r = ttsRead(nInstance, NULL, TTSBUFFER,TTSDEFAULT,TTSDEFAULT,
                  TTSSLICE);
      if ( r != tts_OK ) {
`           PrintError(r); /* process error some way */
      }
      if(r == tts_OK && ! bTransferred){
          MySleep(100);
      }
    } /* until not finished */
```

- **Example 2 (same code 1 in C++)**

```
try {
      While (!i.ttsDone()) {
            i.Read(NULL,TTSBUFFER,TTSDEFAULT,TTSDEFAULT,TSSLICE);
            i.AudioFreeSpace(&bTransferred);
            if (!bTransfered)
                  MySleep(100);
      }
}
catch(CttsError e) {
      …
}
```

ttsGetError **CttsError::CttsError** (class constructor)		Check whether the previous API call has been successful

Classic C Prototype:	`char Ptts_API_DEFINITION ttsGetError(`
	`    ttsHandleType handle`
	`);`	

C++ Class method:	`CttsError::CttsError();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation.	
Return value:	Error message string after an error occurred. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Status	
Notes:	The handle has to be passed to this function depends on the previous API called – use NULL as the implicit session handle	

- **Description**

Returns an error message string after an error occurred. This API can be called as soon as an error occurred (that is any other API has returned a non zero "TTS_OK" value). Pay attention to specify a valid handle as parameter; that is a handle of a successfully created session, instance, or voice. For instance, if a ttsNewInstance API has returned an error, call ttsGetError use the session handle, not the instance one, which has not yet been created.

In case of failure of ttsNewSession there is no use in calling ttsGetError, because even the session handle is invalid.

Using C++ class programming model, there are two public member variable exposed after class construction:

```
const char *m_ErrorString;
ttsResultType m_ErrorCode;
```

ttsSaveStatus **CttsInstance::SaveStatus**		Save the status of all TTS parameters

Classic C Prototype:	`ttsBoolType tts_API_DEFINITION ttsSaveStatus(`
	`    ttsHandleType hInstance`
	`);`	

C++ Class method:	`void CttsInstance::SaveStatus();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Status	
Notes:		

- **Description**

Save the status of all the TTS parameters (Volume, Speed, Pitch, Volume Range, Speed Range, Pitch Range, MaxParPause, MultiCRPause, MultiSpacePause, ProcessingMode, ReadingMode, Spelling Level, Tagged Text, Speaker, Secondary Language, Language Set For Guesser, Input Text Coding, Input Text Type, Default Number Type, etc); it has a “snapshot” behaviour: at the next “ttsSaveStatus”, the previous status will be lost.

ttsRecallStatus **CttsInstance::RecallStatus**		Recall the status of all TTS parameters

Classic C Prototype:	`ttsBoolType tts_API_DEFINITION ttsRecallStatus(`
	`    ttsHandleType hInstance`
	`);`	

C++ Class method:	`void CttsInstance::RecallStatus();`	
Arguments:	`ttsHandleType hInstance` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Status	
Notes:		

- **Description**

Recall the status of all the TTS parameters (Volume, Speed, Pitch, Volume Range, Speed Range, Pitch Range, MaxParPause, MultiCRPause, MultiSpacePause, ProcessingMode, ReadingMode, Spelling Level, Tagged Text, Speaker, Secondary Language, Language Set For Guesser, Input Text Coding, Input Text Type, Default Number Type, etc) at the time of a previous "ttsSaveStatus".

7 Configuration functions

These APIs manage TTS configuration and parameters (audio formats, reading modes, etc.)

ttsLoadConfigurationParam		Loads a keyword value from IniFile or Registry (Win32)

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsLoadConfigurationParam(`
	`    const char * area,`
	`    const char * key,`
	`    ttsInfoStringType value`
	`);`	

C++ Class method:	none	

Arguments:	`const char * area [IN]`
	`const char * key [IN]`
	`ttsInfoStringType value [OUT]`
		Ini file name or registry section [Win32]
		Keyword to load
		Retrieved value

Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Can be used for both session and instance configuration values	

- **Description**

Load a keyword from IniFile or registry (Windows). See chapter 13 for details

ttsSaveConfigurationParam		Saves a keyword value to IniFile or Registry (Win32)

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSaveConfigurationParam(`
	`    const char * area,`
	`    const char * key,`
	`    const char * value`
	`);`	

C++ Class method:	none	
Arguments:	`const char * area` **[IN]** `const char * key` **[IN]** `const char * value` **[IN]**	Ini file name or registry section [Win32] Keyword to save value for the keyword
C++ Class differences	none	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Can be used for both session and instance configuration values	

- **Description**

Save a keyword to IniFile or registry (Windows), creating the IniFile or registry section (if necessary). See chapter 13 for details

ttsDeleteConfigurationParam		Remove a IniFile or registry section

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsDeleteConfiguration(`
	`    const char * area`
	`);`	

C++ Class method:	none	
Arguments:	`const char * area` **`[IN]`**	Ini file name or registry section [Win32]
C++ Class differences	none	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error c ode	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Can be used for both session and instance configuration IniFile or registry section	

- **Description**

Remove the specified IniFile/registry section (area) from disk / registry (Windows). See chapter 13 for details. This API can be used to remove temporary IniFiles or registry sections, created by ttsSaveConfigurationParam

- **Example 1**

```
ttsSaveConfigurationParam(“myIniFile.session”,”LoadingMode”,”RAM”);
r = ttsNewSession(&hSession,”myIniFile.session”);
….
(void)ttsDeleteSession(hSession);
ttsDeleteConfiguration(“myIniFile.session”);
```

ttsGetInstanceParam **CttsInstance::GetParam**		Gets the value of a configuration parameter for current instance

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetInstanceParam(`
	`    ttsHandleType hInstance,`
	`    const char * paramName,`
	`    ttsInfoStringType paramValue`
	`);`	

C++ Class method:	`void GetParam(`
	`    const char * paramName,`
	`    ttsInfoStringType paramValue`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]** `const char * paramName` **[IN]** `ttsInfoStringType paramValue` **[OUT]**	instance handle keyword value to retrieve
C++ Class differences	Instance handle not used in class implementation.	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Retrieve current configuration instance parameters	

- **Description**

See chapter 13 for a list of retrievable values.

- **Example**

```
ttsInfoStringType mode;
ttsGetInstanceParam(hInstance,”ReadingMode”,mode);
printf(“Current reading mode is %s\n”,mode);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsInstance i;
      ttsInfoStringType mode;
      i.GetParam(”ReadingMode”, mode);
      cout << “Current reading mode is” << mode << "\n";
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsSetInstanceParam **CttsInstance::SetParam**		Sets the value of a configuration parameter for current instance

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSetInstanceParam(`
	`ttsHandleType hInstance,`
	`const char * paramName,`
	`const char * paramValue`
	`);`	

C++ Class method:	`void CttsInstance::SetParam(`
	`const char * paramName,`
	`ttsInfoStringType paramValue`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]** `const char * paramName` **[IN]** `ttsInfoStringType paramValue` **[IN]**	instance handle keyword value to Set
C++ Class differences	Instance handle not used in class implementation.	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Set a configuration parameters for current instance (without saving it to IniFile or registry)	

- **Description**

See chapter 13 (13.2) for a list of possible values and keywords.

- **Example 1**

```
ttsSetInstanceParam(hInstance,”ReadingMode”,”ssml”);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsInstance i;
      i.SetParam(”ReadingMode”, “ssml”);
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsGetSessionParam **CttsSession::GetParam**		Gets the value of a configuration parameter for current session

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetSessionParam(`
	`    ttsHandleType hSession,`
	`    const char * paramName,`
	`    ttsInfoStringType paramValue`
	`);`	

C++ Class method:	`void CttsSession::GetParam(`
	`    const char * paramName,`
	`    ttsInfoStringType paramValue`
	`);`	

Arguments:	`ttsHandleType hSession` **[IN]**	session handle
	`const char * paramName` **[IN]**	Keyword
	`ttsInfoStringType paramValue` **[IN]**	value to retrieve
C++ Class differences	Session handle not used in class implementation.	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Retrieve current configuration session parameters. Use NULL as the implicit session handle	

- **Description**

See chapter 13 (13.1) for a list of retrievable values.

- **Example 1**

```
ttsInfoStringType DataPath;
ttsGetSessionParam(hSession,”DataPath”,DataPath);
printf(“Current data path is %s\n”, DataPath);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsSession s;
      ttsInfoStringType DataPath;
      s.GetParam(”DataPath”, DataPath);
      cout << “Current data path is ” << DataPath << "\n";
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsGetVersionInfo		Gets the Loquendo TTS version string

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetVersionInfo(`
	`    ttsInfoStringType StrVer`
	`);`	

C++ Class differences	none	
Arguments:	`ttsInfoStringType StrVer` **`[OUT]`**	string to retrieve
C++ Class differences	none	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:		

- **Description**

Gets the Loquendo TTS version string (x.y.z)

- **Example**

```
ttsInfoStringType LTTSversion;

ttsGetVersionInfo(LTTSversion);
printf(“Current Loquendo TTS version is %s\n”, LTTSversion);
```

ttsDescription **CttsSession::GetDescription**		Gets the description string of a voice

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetDescription(`
	`    ttsHandleType hSession,`
	`    const char *Speaker,`
	`    ttsInfoStringType description`
	`);`	

C++ Class method:	`void CttsSession::GetDescription(`
	`    const char *Speaker,`
	`    ttsInfoStringType description`
	`);`	

Arguments:	`ttsHandleType hSession` **`[IN]`**	session handle
	`const char * Speaker` **`[IN]`**	speaker name
	`ttsInfoStringType description` **`[OUT]`**	value to retrieve
C++ Class differences	Session handle not used in class implementation.	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Does not require that the voice has been opened. Use NULL as the implicit session handle.	

- **Description**

Get the string description of a specified voice (e.g., the description for “Susan” is “American English female voice”)

- **Example 1**

```
ttsInfoStringType Desc;
ttsGetDescription(hSession,”Susan”,Desc);
printf(“Voice: Susan (%s)\n”, Desc);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsSession s;
      ttsInfoStringType Desc;
      s.GetDescription(”Susan”, Desc);
      cout << “Voice: Susan ” << Desc << "\n";
}
catch(CttsError e) {
      cout << "Error no. " << e.m_ErrorCode << "\tMessage: " <<
            e.m_ErrorString << "\n";
}
```

ttsSpeakerLanguage **CttsSession:: SpeakerLanguage**		Returns the language of a voice

Classic C Prototype:	`unsigned int tts_API_DEFINITION ttsSpeakerLanguage(`
	`    ttsHandleType hSession,`
	`    const char *Speaker,`
	`    ttsInfoStringType Language,`
	`    ttsInfoStringType SubLanguage`
	`);`	

C++ Class method:	`unsigned int CttsSession::SpeakerLanguage(`
	`    const char *Speaker,`
	`    ttsInfoStringType Language,`
	`    ttsInfoStringType SubLanguage`
	`);`	

Arguments:	`ttsHandleType hSession` **[IN]** `const char * Speaker` **[IN]** `ttsInfoStringType Language` **[OUT]** `ttsInfoStringType SubLanguage` **[OUT]**	session handle or NULL keyword language string to retrieve sublanguage string to retrieve
C++ Class differences	Session handle not used in class implementation.	
Return value:	Windows Language Identifier (langid) for the language spoken by a specified voice. Use NULL as the implicit session handle. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Does not require that the voice has been opened. Use NULL as the implicit session handle.	

- **Description**

Returns the language spoken by a voice. The information is returned both in the numerical form of the Windows Language Identifier (“langid”) – e.g. 1034 for Castilian Spanish – and in the form of two strings (“language” – e.g. “Spanish” and “sublanguage” e.g. “Castilian”)

- **Example 1**

```
ttsInfoStringType Language,SubLanguage;
ttsSpeakerLanguage(hSession,”Susan”,Language,SubLanguage);
printf(“Language spoken by Susan is %s %s\n”,Language,SubLanguage);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsSession s;
      ttsInfoStringType Language,SubLanguage;
      s.SpeakerLanguage(”Susan”, Language, SubLanguage);
      cout << “Language spoken by Susan is ” << Language <<
                  SubLanguage << "\n";
}
catch(CttsError e) {
…
}
```

ttsGetLanguage **CttsInstance::GetLanguage**		Gets the value of a configuration parameter for current session

Classic C Prototype:	`unsigned int tts_API_DEFINITION ttsGetLanguage(`
	`    ttsHandleType hInstance,`
	`    ttsInfoStringType Language,`
	`    ttsInfoStringType SubLanguage`
	`);`	

C++ Class method:	`unsigned int CttsInstance::GetLanguage(`
	`    ttsInfoStringType Language,`
	`    ttsInfoStringType SubLanguage`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]** `ttsInfoStringType Language` **[OUT]** `ttsInfoStringType SubLanguage` **[OUT]**	Instance handle or NULL language string to retrieve sublanguage string to retrieve
C++ Class differences	Instance handle not used in class implementation.	
Return value:	Windows Language Identifier (langid) for the language spoken by a specified voice. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Equivalent to a combination of ttsSpeakeLanguage and ttsGetActiveVoice	

- **Description**

Returns the language spoken by a voice. The information is returned both in the numerical form of the Windows Language Identifier (“langid”) – e.g. 1034 for Castilian Spanish – and in the form of two strings (“language” – e.g. “Spanish” and “sublanguage” e.g. “Castilian”)

ttsTestVoice **CttsSession::TestVoice**		Tests whether a voice is available

Classic C Prototype:	`ttsBoolType tts_API_DEFINITION ttsTestVoice(`
	`ttsHandleType hSession,`
	`const char * Speaker,`
	`unsigned int SampleRate,`
	`const char * Coding`
	`);`	

C++ Class method:	`ttsBoolType CttsSession::TestVoice(`
	`const char * Speaker,`
	`unsigned int SampleRate,`
	`const char * Coding`
	`);`	

Arguments:	`ttsHandleType hSession` **`[IN]`**	session handle or NULL
	`const char * Speaker` **`[IN]`**	speaker name
	`unsigned int SampleRate` **`[IN]`**	sample rate in hz
	`const char *Coding` **`[IN]`**	sample coding { "L" (linear), "A" (A-law), "U" (u-law)
C++ Class differences	Session handle not used in class implementation.	
Return value:	ttsTRUE if the voice is installed, ttsFALSE if not. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	Use NULL as the implicit session handle.	

- **Description**

Tests whether a voice (combination of Speaker, Sample Rate and coding) is available (that is has been installed)

ttsGetActiveVoice **CttsInstance::GetActiveVoice**		Retrieve current instance voice

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetActiveVoice(`
	`    ttsHandleType hInstance,`
	`    ttsInfoStringType Speaker,`
	`    unsigned int *SampleRate,`
	`    ttsInfoStringType Coding`
	`);`	

C++ Class method:	`void CttsInstance::GetActiveVoice(`
	`    ttsInfoStringType Speaker,`
	`    unsigned int *SampleRate,`
	`    ttsInfoStringType Coding`
	`);`	

Arguments:	`ttsHandleType hInstance` **`[IN]`**	session handle
	`const char * Speaker` **`[OUT]`**	speaker name (to retrieve)
	`unsigned int *SampleRate` **`[OUT]`**	sample rate in hz (to retrieve)
	`ttsInfoStringType Coding` **`[OUT]`**	sample coding { “L” (linear), “A” (A-law), “U” (u-law) } to retrieve
C++ Class differences	Instance handle not used in class implementation.	
Return value:	TTS_OK (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	
Notes:	2nd , 3rd and/or 4th arguments may be NULL	

- **Description**

Retrieve current instance voice (combination of Speaker, Sample Rate and coding)

- **Example 1**

```
ttsInfoStringType speaker;
ttsGetActive(hInstance,speaker,NULL,NULL);
printf(“Current speaker is %s\n”, speaker);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsInstance i;
      ttsInfoStringType Speaker;
      i.GetActiveVoice speaker,NULL,NULL);
      cout << “Current speaker is ” << Speaker << "\n";
}
catch(CttsError e) {
      …
}
```

ttsQuery **CttsInstance::Query** **CttsSession::Query**		Lists available voices and gets all their parameters

Classic C Prototype:	unsigned int tts_API_DEFINITION ttsQuery(
	    ttsHandleType Handle,
	    const char *RequestedData,
	    const char *DataConditions,
	    char *QueryResult,
	    unsigned int QueryResultLen,
	    ttsBoolType bRescanFileSystem
	);	

C++ Class method:	unsigned int CttsInstance::Query(
	    const char *RequestedData,
	    const char *DataConditions,
	    char *QueryResult,
	    unsigned int QueryResultLen,
	    ttsBoolType bRescanFileSystem
	);
	unsigned int CttsSession::Query(
	    const char *RequestedData,
	    const char *DataConditions,
	    char *QueryResult,
	    unsigned int QueryResultLen,
	    ttsBoolType bRescanFileSystem
	);	

Arguments:	ttsHandleType Handle **[IN]**	Session/instance handle
	const char *RequestedData **[IN]**	requested data (e.g. "Speaker")
	const char *DataConditions **[IN]**	comma separated list of conditions (e.g. "Frequency = 16000, Coding = L")
	char *QueryResult **[OUT]**	semicolon separated list of values (e.g. "Susan; Dave")
	unsigned int QueryResultLen [IN]	size of QueryResult in bytes
	ttsBoolType bRescanFileSystem [IN]	True → Rescan the file system and rebuild data structures for all installed voices (requires a session handle as 1st parameter)
C++ Class differences	Instance/Session handle not used in class implementation.	
Return value:	# of returned values. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Configuration	

Notes:	Depending on the handle passed as 1st argument (instance or session), the query is performed on current instance (that is all open voices) or on the entire session (all installed voices). Use NULL as the implicit session handle.

- **Description**

This complex API allows making interactive queries on the open and/or installed voices, retrieving all voice parameters (such as Language, SubLanguage, gender) and/or search for voices based on specific parameters (e.g. enumerate all 16 khz linear German voices – see the example below).

The keys that can be used both in RequestData and DataConditions arguments are the following: Speaker (the voice name), Description, Language, SubLanguage, Frequency (only in DataCondition), Coding(only in DataCondition), Gender, Age, WinLangID (used in Windows applications), FrequencyAndCoding (only in RequestData; a voice is specified by the couple of frequency and coding parameters, so a “frequency.coding” value is returned).

The format of the RequestData string must be a string containing one ore more of the previous keys separated by commas (i.e. “Speaker, Gender, Age”), while the format of the DataConditions is a bit more complex, because every key must be followed by the condition (i.e “Speaker=Mario, Gender=male”).

Inside the result string, named QueryResult, there will be a list of values separated by semicolons; every value is correspondent to a RequestData string, with the same colons separated syntax, but the number of values can be variable, according to the conditions imposed by DataConditions and to the installation situation.

If RequestData =“Speaker, Gender” and DataCondition = “Language=English”, you could obtain something like this: “Susan, female; Dave, male; Kenneth, male” in QueryResult string.

- **Example 1**

```
char qr[1000];
int n = ttsQuery(hSession,”Speaker”,
   ”Frequency=16000, Coding=L, Language=German”,qr,1000,ttsFALSE);
char *p=strtok(qr,";");
printf(“%d voices found:”,n);
while(p)
{
      printf(“ %s”,p);
      p = strtok(NULL,";");
}
printf(“\n”);
```

- **Example 2 (same code 1 in C++)**

```
try {
      char qr[1000];
      CttsSession s;
      int n = s.Query(”Speaker”, ”Frequency=16000, Coding=L,
                      Language=German”,qr,1000,ttsFALSE);
      …
}
catch(CttsError e) {
      …
}
```

8 Prosody functions

ttsSetPitch **CttsInstance::SetPitch**		Changes Pitch baseline

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSetPitch(`
	`ttsHandleType hInstance`
	`unsigned int value`
	`);`	

C++ Class method:	`void CttsInstance::SetPitch(`
	`unsigned int value`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]** `unsigned int value` **[IN]**	instance handle pitch baseline
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Changes the pitch baseline – the default range is 0-100. One can change the pitch range by calling ttsSetPitchRange. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

- **Example 1**

```
ttsSetSpeed(hInstance,speed);
ttsSetPitch(hInstance,pitch);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsInstance i;
      i.SetSpeed(8);
      …
}
catch(CttsError e) {
      …

}
```

See Also ttsGetPitch, ttsSetPitchRange, ttsSetDefaultAttributes

ttsGetPitch **CttsInstance::GetPitch**		Get the pitch baseline

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetPitch(`
	`      ttsHandleType hInstance`
	`      unsigned int *value`
	`);`	

C++ Class method:	`void CttsInstance::GetPitch(`
	`      unsigned int *value`
	`);`	

Arguments:	`ttsHandleType Handle` **`[IN]`**
	`unsigned int *value` **`[OUT]`**
		instance handle
		pitch baseline

C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Gets the pitch baseline – the default range is 0-100. One can change the pitch range by calling ttsSetPitchRange. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

See Also ttsSetPitch, ttsSetPitchRange, ttsSetDefaultAttributes

ttsSetSpeed **CttsInstance::SetSpeed**		Speaking rate setting

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSetSpeed(`
	`      ttsHandleType hInstance`
	`      unsigned int value`
	`);`	

C++ Class method:	`void CttsInstance::SetSpeed (`
	`      unsigned int value`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]** `unsigned int value` **[IN]**	instance handle speed
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Changes the speed – the default range is 0-100. One can change the speed range by calling ttsSetSpeedRange. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

- **See Also ttsGetSpeed, ttsSetSpeedRange, ttsSetDefaultAttributes**

ttsGetSpeed CttsInstance::GetSpeed		Retrieve current speaking rate

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetSpeed(`
	`      ttsHandleType hInstance`
	`      unsigned int *value`
	`);`	

C++ Class method:	`void CttsInstance::GetSpeed(`
	`      unsigned int *value`
	`);`	

Arguments:	`ttsHandleType Handle` **`[IN]`**
	`unsigned int *value` **`[OUT]`**
		instance handle
		speed

C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Gets the pitch baseline – the default range is 0-100. One can change the speed range by calling ttsSetSpeedRange. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

- **See Also ttsSetSpeed, ttsSetSpeedRange, ttsSetDefaultAttributes**

ttsSetVolume **CttsInstance::SetVolume**		Set current volume

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSetVolume(`
	`ttsHandleType hInstance`
	`unsigned int value`
	`);`	

C++ Class method:	`void CttsInstance::SetVolume(`
	`unsigned int value`
	`);`	

Arguments:	`ttsHandleType Handle` **`[IN]`**
	`unsigned int value` **`[IN]`**
		instance handle
		volume

C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Changes the volume. The default range is 0-100. One can change the volume range by calling ttsSetVolumeRange. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

- **See Also**

ttsGetVolume, ttsSetDefaultAttributes, ttsSetVolumeRange

ttsGetVolume **CttsInstance::GetVolume**		Retrieve current volume

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetVolume(`
	`      ttsHandleType hInstance`
	`      unsigned int *value`
	`);`	

C++ Class method:	`void CttsInstance::GetVolume(`
	`      unsigned int *value`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]** `unsigned int *value` **[OUT]**	instance handle volume
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Gets the volume of the voice in use. The default range is 0-100. One can change the volume range by calling ttsSetVolumeRange. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

See Also ttsSetVolume, ttsSetDefaultAttributes

ttsSetDefaultAttributes **CttsInstance:: SetDefaultAttributes**		Assigns the default value to all prosody attributes

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSetDefaultAttributes(`
	`      ttsHandleType hInstance`
	`);`	

C++ Class method:	`void CttsInstance::SetDefaultAttributes();`	
Arguments:	`ttsHandleType Handle` **`[IN]`**	instance handle
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Resets all prosody attributes (speed, pitch, volume). However, it does not reset user defined speed/pitch/volume ranges.

- **See Also ttsSetVolume, ttsSetPitch, ttsSetSpeed**

ttsSetPitchRange **CttsInstance::SetPitchRange**		Assigns values to the pitch range

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSetPitchRange(`
	`ttsHandleType hInstance`
	`unsigned int minimum,`
	`unsigned int normal,`
	`unsigned int maximum`
	`);`	

C++ Class method:	`void CttsInstance::SetPitchRange(`
	`unsigned int minimum,`
	`unsigned int normal,`
	`unsigned int maximum`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]**	instance handle
	`unsigned int minimum` **[IN]**	Minimum value of the pitch range
	`unsigned int normal` **[IN]**	Normal value of the pitch range
	`unsigned int maximum` **[IN]**	Maximum value of the pitch range
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Assigns values to the pitch range (the default range is 0-50-100) Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

- **Example1**

```
ttsSetPitchRange(hInstance,0,10,20);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsInstance i;
      i.SetPitchRange(0,10,20);
      …
}
catch(CttsError e) {
      …
}
```

ttsSetVolumeRange **CttsInstance::SetVolumeRange**		Assigns values to the volume range

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSetVolumeRange(`
	`ttsHandleType hInstance`
	`unsigned int minimum,`
	`unsigned int normal,`
	`unsigned int maximum`
	`);`	

C++ Class method:	`void CttsInstance::SetVolumeRange(`
	`unsigned int minimum,`
	`unsigned int normal,`
	`unsigned int maximum`
	`);`	

Arguments:	`ttsHandleType Handle` **`[IN]`**	instance handle
	`unsigned int minimum` **`[IN]`**	Minimum value of the volume range
	`unsigned int normal` **`[IN]`**	Normal value of the volume range
	`unsigned int maximum` **`[IN]`**	Maximum value of the volume range
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Assigns values to the volume range. The default value is 0, 50, 100. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

- **Example 1**

```
ttsSetVolumeRange(hInstance,0,5,10);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsInstance i;
      i.SetVolumeRange(hInstance,0,5,10);
      …
}
catch(CttsError e) {
      …

}
```

ttsSetSpeedRange **CttsInstance::SetSpeedRange**		Assigns values to the Speed range

C/C++ Prototype:	`ttsResultType tts_API_DEFINITION ttsSetSpeedRange (`
	`      ttsHandleType hInstance`
	`      unsigned int minimum,`
	`      unsigned int normal,`
	`      unsigned int maximum`
	`);`	

C++ Class method:	`void CttsInstance::SetSpeedRange(`
	`      unsigned int minimum,`
	`      unsigned int normal,`
	`      unsigned int maximum`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]**	instance handle
	`unsigned int minimum` **[IN]**	Minimum value of the speed range
	`unsigned int normal` **[IN]**	Normal value of the speed range
	`unsigned int maximum` **[IN]**	Maximum value of the speed range
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Assigns values to the speed range (the default range is 0-50-100). Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

- **Example 1**

```
ttsSetSpeedRange(hInstance,0,10,20);
```

- **Example 2 (same code 1 in C++)**

```
try {
      CttsInstance i;
      i.SetSpeedRange(hInstance,0,10,20);
      …
}
catch(CttsError e) {
      …
}
```

ttsGetPitchRange **CttsInstance::GetPitchRange**		Get the values of the pitch range

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetPitchRange(`
	`      ttsHandleType hInstance`
	`      unsigned int *minimum,`
	`      unsigned int *normal,`
	`      unsigned int *maximum`
	`);`	

C++ Class method:	`void CttsInstance::GetPitchRange(`
	`      unsigned int *minimum,`
	`      unsigned int *normal,`
	`      unsigned int *maximum`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]** `unsigned int *minimum` **[OUT]** `unsigned int *normal` **[OUT]** `unsigned int *maximum` **[OUT]**	instance handle Minimum value of the pitch range Normal value of the pitch range Maximum value of the pitch range
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherw ise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Get the values of the pitch range (the default range is 0-50-100) Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

ttsGetVolumeRange **CttsInstance::GetVolumeRange**		Get the values of the volume range

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetVolumeRange(`
	`ttsHandleType hInstance`
	`unsigned int *minimum,`
	`unsigned int *normal,`
	`unsigned int *maximum`
	`);`	

C++ Class method:	`void CttsInstance::GetVolumeRange(`
	`unsigned int *minimum,`
	`unsigned int *normal,`
	`unsigned int *maximum`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]**	instance handle
	`unsigned int minimum` **[OUT]**	Minimum value of the volume range
	`unsigned int normal` **[OUT]**	Normal value of the volume range
	`unsigned int maximum` **[OUT]**	Maximum value of the volume range
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Get the values of the volume range. The default value is 0, 50, 100. Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

ttsGetSpeedRange **CttsInstance::GetSpeedRange**		Get the values of the Speed range

C/C++ Prototype:	`ttsResultType tts_API_DEFINITION ttsGetSpeedRange (`
	`ttsHandleType hInstance`
	`unsigned int *minimum,`
	`unsigned int *normal,`
	`unsigned int *maximum`
	`);`	

C++ Class method:	`void CttsInstance::GetSpeedRange(`
	`unsigned int *minimum,`
	`unsigned int *normal,`
	`unsigned int *maximum`
	`);`	

Arguments:	`ttsHandleType Handle` **[IN]**	instance handle
	`unsigned int *minimum` **[OUT]**	Minimum value of the speed range
	`unsigned int *normal` **[OUT]**	Normal value of the speed range
	`unsigned int *maximum` **[OUT]**	Maximum value of the speed range
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

Get the values of the speed range (the default range is 0-50-100). Pay attention: up to the previous 6.3.x versions, the range was 0 to 10; it is possible to restore this behaviour by setting this key: OldProsodyRange=true.

9 Lexicon functions

These APIs manage exception lexicons (expansions, phonetic transcriptions)

ttsNewLexicon **CttsLexicon::CttsLexicon** (class constructor)		Opens a user-lexicon file and attaches to current voice

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsNewLexicon(`
	`    ttsHandleType *hLexicon,`
	`    ttsHandleType hVoice,`
	`    const char *filename`
	`);`	

C++ Class method:	`CttsLexicon:: CttsLexicon(`
	`    CttsVoice *voice,`
	`    const char *filename`
	`);`	

Arguments:	`ttsHandleType *hLexicon` **[OUT]** `ttsHandleType hVoice` **[IN]** `const char * filename` **[IN]**	lexicon handle voice handle full pathname of a user-lexicon file
C++ Class differences	In place of the voice handle, the class constructor method uses a pointer to a CttsVoice class previously created, and the pointer to the CttsLexicon class created is equivalent to the lexicon handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:		

- **Description**

This function sets up a user-lexicon file (attaching it to current voice), containing phonetic transcriptions of words and expansions (e.g. Mr. = mister). For information on how to compile or edit a lexicon file, refer to the "Loquendo TTS User Guide". More than one user-lexicon can be opened. Filename can be a valid URL too (supported on Windows, on Linux by means of the library "libcurl.so" usually included in the Linux distributions, not supported on Solaris).

- **Example 1**

```
ttsNewLexicon(&hLexicon,hVoice,“c:\\loqtts\\data\\custom.lex”);
```

- **Example 2 (same code 1 in C++)**

```
try {
      // Implicit Instance creation
      CttsVoice v(NULL, ”Susan”, 16000, “a”);
      // Setting-up User Lexicon
      CttsLexicon l(&v, “c:\\loqtts\\data\\custom.lex”);
      …
}
catch(CttsError e) {
      …
}
```

ttsDeleteLexicon **CttsLexicon::~CttsLexicon** (class destructor)		Closes a lexicon, detaching it from current voice

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsDeleteLexicon(`
	`    ttsHandleType hLexicon`
	`);`	

C++ Class method:	`void CttsLexicon::~CttsLexicon();`	
Arguments:	`ttsHandleType hLexicon` **`[IN]`**	lexicon handle
C++ Class differences	CttsLexicon class desctructor method: no paraterers needs	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:	Removes the user-lexicon from current voice but it does not delete lexicon file from disk.	

- **Description**

The function closes a lexicon, detaching it from current voice

- **See also**

ttsNewLexicon

ttsGetLexiconEntry **CttsLexicon::GetEntry**		Retrieve a lexicon transcription

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsGetLexiconEntry(`
	`    ttsHandleType hLexicon,`
	`    const char * string,`
	`    ttsInfoStringType trascription`
	`);`	

C++ Class method:	`void CttsLexicon:: GetEntry(`
	`    const char * string,`
	`    ttsInfoStringType trascription`
	`);`	

Arguments:	`ttsHandleType hLexicon` **[IN]**	lexicon handle
	`const char * string` **[IN]**	String to be searched in the lexicon
	`ttsInfoStringType trascription` **[OUT**	Retrieved transcription string (can be empty if no transcription exists for that string)
C++ Class differences	The pointer to the CttsLexicon class is equivalent to the lexicon handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:	any	

- **Description**

Returns the transcription of a word from of a lexicon, if any. For example: with input "Mr", the function returns the string "mister".

ttsAddLexiconEntry CttsLexicon::AddEntry		Adds an entry to a lexicon

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsAddLexiconEntry(`
	`    ttsHandleType hLexicon,`
	`    const char * string,`
	`    const char * transcription`
	`);`	

C++ Class method:	`void CttsLexicon::AddEntry(`
	`    const char * string,`
	`    const char * transcription`
	`);`	

Arguments:	`ttsHandleType hLexicon` **`[IN]`**	lexicon handle
	`const char * string` **`[IN]`**	String to be searched in the lexicon
	`const char *transcription` **`[IN]`**	Transcription to add
C++ Class differences	The pointer to the CttsLexicon class is equivalent to the lexicon handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:	Entries added are not saved into lexicon file until the application calls explicitly ttsSaveLexicon	

- **Description**

Adds a new entry to a lexicon.

- **Example 1**

```
RetVal = ttsAddLexiconEntry(hLexicon,”Mr”,”mister”)
```

- **Example 2 (same code 1 in C++)**

```
try {
      // Implicit Instance creation
      CttsVoice v(NULL, ”Susan”, 16000, “a”);
      // Setting-up User Lexicon
      CttsLexicon l(&v, “c:\\loqtts\\data\\custom.lex”);
      l.AddEntry(”Mr”,”mister”);
      …
}
catch(CttsError e) {
      …

}
```

- **See Also**

RemoveLexiconEntry, ttsSaveLexicon

ttsRemoveLexiconEntry CttsLexicon::RemoveEntry		Removes an entry from a lexicon

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsRemoveLexiconEntry(`
	`    ttsHandleType hLexicon,`
	`    const char * string`
	`);`	

C++ Class method:	`void CttsLexicon::RemoveEntry(`
	`    const char * string,`
	`);`	

Arguments:	`ttsHandleType hLexicon` **[IN]** `const char * string` **[IN]**	lexicon handle Entry to remove
C++ Class differences	The pointer to the CttsLexicon class is equivalent to the lexicon handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:	Entries are not removed from lexicon file until the application calls explicitly ttsSaveLexicon	

- **Description**

Removes an entry from a lexicon

- **See Also**

AddLexiconEntry, ttsSaveLexicon

- **Example 1**

```
RetVal = ttsRemoveLexiconEntry(hLexicon,”Mr.”);
```

- **Example 2 (same code 1 in C++)**

```
try {
      // Implicit Instance creation
      CttsVoice v(NULL, ”Susan”, 16000, “a”);
      // Setting-up User Lexicon
      CttsLexicon l(&v, “c:\\loqtts\\data\\custom.lex”);
      l.AddEntry(”Mr”,”mister”);
      …
      l.RemoveEntry(”Mr.”);
      …
}
catch(CttsError e) {
      …

}
```

ttsSaveLexicon **CttsLexicon::Save**		Saves a lexicon

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsSaveLexicon(`
	`    ttsHandleType hLexicon,`
	`    const char * filename`
	`);`	

C++ Class method:	`void CttsLexicon::Save(`
	`    const char * filename`
	`);`	

Arguments:	`ttsHandleType hLexicon` **[IN]** `const char * filename` **[IN]**	lexicon handle Lexicon file name
C++ Class differences	The pointer to the CttsLexicon class is equivalent to the lexicon handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Prosody	
Notes:	Creates lexicon file if does not exist	

- **Description**

Saves a programmatically modified lexicon file to disk. Filename can be a valid URL too (supported on Windows, on Linux by means of the library “libcurl.so” usually included in the Linux distributions, not supported on Solaris).

10 Utility functions

These APIs export methods for managing phonemes, XML and text

ttsPhoneticTranscription **CttsInstance::PhoneticTranscription**		Return the phonetic transcription of a text string.

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsPhoneticTranscription(`
	`    ttsHandleType hInstance,`
	`    const void * GraphemeString,`
	`    void *PhonemeString,`
	`    unsigned long *duration,`
	`    int TextCoding,`
	`    int ReadingMode`
	`);`	

C++ Class method:	`void CttsInstance::PhoneticTranscription(`
	`    const void * GraphemeString,`
	`    void *PhonemeString,`
	`    unsigned long *duration,`
	`    int TextCoding,`
	`    int ReadingMode`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]** `const void * GraphemeString` **[IN]** `void *PhonemeString` [**OUT**] `unsigned long *duration` **[OUT]** `int TextCoding` **[IN]** `int ReadingMode` **[IN]**	instance handle Text to transcribe Phonetic transcription (ASCII) Duration in ms TTSANSI, TTSUNICODE, etc TTSSSML, TTSMULTILINE, etc
C++ Class differences	The pointer to the CttsInstance class is equivalent to the instance handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Utility	
Notes:	For a description of 5th and 6th arguments, see ttsRead documentation	

- **Description**

ttsPhoneticTranscription returns the phonetic transcription of a text string. The phonemes used are described in the Loquendo TTS Language Reference Guides and may vary depending on the language.

ttsCheckPhoneticTranscription **CttsInstance::CheckPhoneticTranscription**		Check the syntax of a phonetic transcription

Classic C Prototype:	ttsResultType tts_API_DEFINITION
	ttsCheckPhoneticTranscription(
	    ttsHandleType hInstance,
	    const char *PhoneticString
	);	

C++ Class method:	void CttsInstance:: CheckPhoneticTranscription(
	    ttsHandleType hInstance,
	    const char *PhoneticString
	);	

Arguments:	ttsHandleType hInstance **[IN]** void * PhoneticString [**IN**]	instance handle Phonetic string to check
C++ Class differences	The pointer to the CttsInstance class is equivalent to the instance handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Utility	
Notes:	requires a valid open instance	

- **Description**

ttsCheckPhoneticTranscription returns an error code when the syntax of a phonetic string is incorrect. It may be used for debugging purposes.

ttsPhoneticUtils		Manages phonetic symbols in many ways.

Classic CPrototype:	`unsigned long ttsPhoneticUtils (`
	`      unsigned long utility,`
	`      ...`
	`);`	

C++ Class method:	none	

Arguments:	`unsigned long utility` **`[IN]`**
	`< list of arguments >` **`[IN/OUT]`**
		Phonetic utility identifier: can be:
		TTS_PHONEMEMNEMONIC
		TTS_GETPHONETICSTREAM
		TTS_LOQUENDOPHONEME2IPA
		TTS_IPASTRING2IPACODE
		TTS_GETPHONEMESNUMBER
		TTS_IPACODE2IPASTRINGREDUCED
		TTS_REASONABLEPHOMNEMONIC
		Optional arguments depending on the chosen utility (see description below)

C++ Class differences	none	
Return:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code	
Inclusions:	loqtts.h	
Category:	Utility	
Notes:	This is a variable arguments function	

- **Description**

ttsPhoneticUtils is a *variadic* function which can handle many different arguments depending on the utility the caller invoke by the *utility* parameter. You don't have to write anything special when you call this function. Just pass the utility identifier, the optional parameters listed below and cast the result according to the return type handled by the utility.

- **TTS_PHONEMEMNEMONIC**: returns the phonetic symbol of a phoneme number. The phonemes used are described in the Loquendo TTS user manual and may vary depending on the language. The arguments are:

ttsHandleType hInstance	hInstance is a synthesis's instance
unsigned int PhonemeNumber	The phoneme number.

Returns the `(const char*)` phonetic symbol in case of success, NULL otherwise.

- **Example**

```
const char* sym;
sym=(const
char*)ttsPhoneticUtils(TTS_PHONEMEMNEMONIC,hInstance,PhoNum);
printf(“%s\n”,sym);
```

- **TTS_GETPHONETICSTREAM**: this utility allows to get the phoneme number, duration and position in the phonetic stream. The arguments are:

unsigned long handle	The handle containing the phonetic stream
unsigned int* ph	The phoneme number returned by the function
unsigned int* dur	The phoneme duration returned by the function
unsigned int* pos	The position into the phonetic stream returned by the function

Returns (unsigned int)ttsFALSE when the end of the phonetic stream is reached, ttsTRUE otherwise.

- **Example:**

```
unsigned int ph,dur,pos;
while((unsigned int)ttsPhoneticUtils(
      TTS_GETPHONETICSTREAM,handle,&ph,&dur,&pos)==ttsTRUE{
      printf(“Phoneme n. %d – duration %d – position %d\n”,
      ph,dur,pos);
}
```

- **TTS_LOQUENDOPHONEME2IPA**: This utility convert the phoneme symbols used in LoquendoTTS (described in the Loquendo TTS user manual) to the IPA representation. The arguments are:

ttsHandleType hInstance	hInstance is a synthesis’s instance
wchar_t* ipastring	The ipa representation returned by the function
ttsInfoStringType phoneme	The phoneme to convert

Returns always tts_OK.

- **Example:**

```
wchar_t ipastring[MAXLENGTH];
(ttsResultType)ttsPhoneticUtils(TTS_LOQUENDOPHONEME2IPA,
                                hInst,ipa,lttsp);
```

- **TTS_GETPHONEMESNUMBER**: This utility returns the number of phonemes of the language. The arguments are:

ttsHandleType hInstance	hInstance is a synthesis’s instance

Returns the (unsigned int) number of phonemes.

- **Example:**

```
unsigned int PhoNum;
PhoNum = ttsPhoneticUtils(TTS_GETPHONEMESSNUMBER,hInst);
```

- **TTS_REASONABLEPNONEMEMNREMONIC**: This utility returns the most similar phoneme to the given phoneme if the given phoneme doesn’t exist, the exact phoneme otherwise. The arguments are:

ttsHandleType hInstance	hInstance is a synthesis’s instance
unsigned int PhonemeNumber	The phoneme number.

Returns the `(const char*)` the phoneme string.

- **Example**

```
const chat* pho;
pho = (const char*)
ttsPhoneticUtils(TTS_REASONABLEPHOMNEMONIC,PhoNum);
```

ttsLanguageGuess **CttsInstance::LanguageGuess**		Detects the language of a chunk of text

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsLanguageGuess(`
	`    ttsHandleType hInstance,`
	`    const void *Input,`
	`    int InputType,`
	`    int TextCoding,`
	`    int ReadingMode,`
	`    ttsInfoStringType GuessedLanguage`
	`);`	

C++ Class method:	`void CttsInstance::LanguageGuess(`
	`    const void *Input,`
	`    int InputType,`
	`    int TextCoding,`
	`    int ReadingMode,`
	`    ttsInfoStringType GuessedLanguage`
	`);`	

Arguments:	`ttsHandleType hInstance` **[IN]**	instance handle
	`const void *Input` **[IN]**	text buffer or file name
	`int InputType` **[IN]**	TTSFILE or TTSBUFFER
	`int TextCoding` **[IN]**	TTSANSI, TTSUNICODE, etc
	`int ReadingMode` **[IN]**	TTSSMAL, TTSMULTINE, etc.
	`ttsInfoStringType GuessedLanguage` **[OUT]**	retrieved language string
C++ Class differences	The pointer to the CttsInstance class is equivalent to the instance handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Utility	
Notes:	Requires “Loquendo Mixed Language Support CD”	

- **Description**

ttsLanguageGuess detects the language in which a specified block of text or file is written. It uses a stochastic method based on character sequences occurrence. The longer the text is, the better it works. For a description of arguments 3, 4, 5 see ttsRead reference. The guessed language string is returned on argument 6. At the moment, possible values are:

ENGLISH, GERMAN, FRENCH, ITALIAN, SPANISH, CATALAN, GREEK, SWEDISH, PORTUGUESE

Passing NULL as second argument (“Input”) fills the “GuessedLanguage” string with a semicolon-separated list of all available languages.

ttsValidateXML **CttsInstance:: ValidateXML**		Checks if a chunk of text is XML well-formed

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsValidateXML(`
	`    ttsHandleType hInstance,`
	`    const void *Input,`
	`    int InputType,`
	`    int TextCoding`
	`);`	

C++ Class method:	`void CttsInstance::ValidateXML(`
	`    const void *Input,`
	`    int InputType,`
	`    int TextCoding`
	`);`	

Arguments:	`ttsHandleType hInstance` **`[IN]`**
	`const void *Input` **`[IN]`**
	`int InputType` **`[IN]`**
	`int TextCoding` **`[IN]`**
		instance handle
		text buffer or file name
		TTSFILE or TTSBUFFER
		TTSANSI, TTSUNICODE, etc

C++ Class differences	The pointer to the CttsInstance class is equivalent to the instance handle.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Utility	
Notes:		

- **Description**

ttsValidateXML checks if a text buffer or file is XML well-formed (that is can be read correctly by Loquendo TTS in TTSSSML mode). For a description of arguments 3 and 4 see ttsRead reference.

ttsClaimLicense **CttsInstance::ClaimLicense**		Reserve a license (if available)

Classic C Prototype:	`ttsResultType tts_API_DEFINITION ttsClaimLicense(`
	`ttsHandleType hInstance`
	`unsigned long *time2wait`
	`);`	

C++ Class method:	`void CttsInstance::ClaimLicense(`
	`unsigned long *time2wait`
	`);`	

Arguments:	`ttsHandleType Handle` **`[IN]`**	instance handle
	`unsigned long *time2wait` **`[OUT]`**	time value to wait (in msec)
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Utilities	
Notes:		

- **Description**

The ttsClaimLicense reserve a license (if available) in a permanent way. In order to realize a guarenteed resource license model, it is sufficient to call the ttsClaimLicense at the instance opening and to call the ttsUnclaimLicense (see) at the instance close.

The instance will have no need to acquire a license at the ttsRead, so if the instance is idle, the ttsRead will be immediately performed.

If no license are available, the return value will be different from tts_OK, and the parameter time2wait will contain the time to wait (in msec) to get the license available.

ttsUnclaimLicense **CttsInstance::UnclaimLicense**		Release a license previously reserved

Classic C Prototype:	ttsResultType tts_API_DEFINITION ttsUnclaimLicense(
	      ttsHandleType hInstance
	      unsigned long *time2wait
	);	

C++ Class method:	void CttsInstance::UnclaimLicense(
	      unsigned long *time2wait
	);	

Arguments:	ttsHandleType Handle **[IN]** unsigned long *time2wait **[OUT]**	instance handle time value to wait (in msec)
C++ Class differences	Instance handle not used in class implementation.	
Return value:	tts_OK, (zero) in case of success. Otherwise returns a 32 bit error code. Use CttsError class for C++ exception handling.	
Inclusions:	loqtts.h	
Category:	Utilities	
Notes:		

- **Description**

The ttsUnclaimLicense release a license reserved (in a permanent way) by a previous tttClaimLicense (see).

After the call to ttsUnclaimLicense, the parameter time2wait will contain the time to wait (in msec) to get the license free again (useful only for statistical purposes). The realese is not immediate, because the availability of the license token is related to the number of samples produced by the TTS conversion.

11 Mixed Language Configuration

Loquendo TTS 6 can guess the language of a chunk of text. This can be made on an application-specific request (see ttsGuessLanguage API), or automatically.

The automatic guessing can be enabled no matter of the API set used (tts or SAPI). Two different modes are possible:

1. Language Switch
2. Voice Switch

In mode 1) the language is automatically changed, without switching the active voice. For instance, the American English voice "Dave" can switch temporarily to French, and use the French rule set, in order to pronounce a French sentence, and then come back to English. The French pronunciation is less accurate than a French voice's one: it sounds more like an English native speaker that speaks French.

In mode 2) the voice is changed automatically, choosing the most appropriate one among the installed voices. In case more than a voice is present, speaking the same language, here is the precedence:

1. Among the open voices (already loaded in memory), finds for a voice of the desiderated language, with the same sex of the currently active voice
2. Among the open voices (already loaded in memory), finds a voice of the desiderated language
3. Finds an installed voice (not already loaded in memory) of the desiderated language, with the same sex of the currently active voice
4. Finds an installed voice (not already loaded in memory) of the desiderated language

If Loquendo TTS cannot find a voice to perform the voice switching, the command is ignored.

The automatic guessing uses the Language Guesser to detect the language; the application must define the length of the part of speech the guessing must be applied to, among:

1. Paragraph by Paragraph
2. Sentence by Sentence
3. Phrase by Phrase
4. Word by Word

"Phrase by Phrase" and "Word by word" modes make sense only combined with the Language Switch, whilst the other two modes can be applied both to Language and Voice Switches.

Note that "Word by word" mode may sometimes lead to unpredictable results, due to intrinsic ambiguity of most words. For instance the sentence "Mission impossible" can be either English or French. The guessing would be more accurate when applied to a longer part of speech.

Finally, in order to facilitate the Language Guesser job, it is possible to define the list of languages to guess among.

See Instance parameters section to know how to enable/configure the automatic guesser

12 Application callback and Loquendo TTS Events

In the asynchronous programming model, it's usually necessary to be notified whenever a relevant event occurs, such as the end of the speech conversion. Without that event notification, for instance, the application wouldn't be aware of the moment in which the system will be ready to perform a new speech synthesis. A GUI application may use asynchronous events to change the appearance of its buttons (for instance a "Speak" button could be grayed as soon as speaking starts and re-enabled at the end of the speech conversion).

Loquendo TTS notifies the application of its events by means of the so-called application callback. The application callback is a user-defined function declared as:

```
typedef void (TTSCALLBACK *ttsCallbackType)(
    ttsEventType nReason,
    long lData,
    void *pUser
);
```

The application informs Loquendo TTS of its callback by calling the ttsRegisterCallback API. For instance:

```
ttsRegisterCallback(hInstance,myCallback,&userdata,0);
```

declares that the application callback function for instance "hInstance" is named "myCallback", and that the pointer to "userdata" must be passed on every call to "myCallback". In this way the user defined function can access application-specific data (Loquendo TTS doesn't know anything about the structure of "userdata", but simply passes a "void *" reference to that). The last parameter of ttsRegisterCallback is reserved for future use.

Somewhere in the application the function

```
void TTSCALLBACK Callback(ttsEventType nReason, void *lData, void *pUser)
```

will be present, either static or not, depending on the implementation.

The first parameter "nReason" is the kind of event that occurred, while "lData" is the event-specific data associated with nReason event. Finally "pUser" contains the application-specific data ("userdata").

Please note the same instance may call the callback function by different threads simultaneously. Therefore it is strictly necessary that the function is reentrant (e.g. avoiding the use of global variables) and/or sensible data is protected by concurrent access (by means of semaphores, mutexes, or critical sections).

Here is a list of the events managed by Loquendo TTS (declared in "loqtts.h"[12]):

Event	Meaning
TTSEVT_TEXT	Text has been received and split into paragraph
TTSEVT_WORDTRANSCRIPTION	A word has been phonetically transcribed
TTSEVT_PHONEMES	A stream of phonemes has been produced
TTSEVT_AUDIOSTART	Audio rendering has started
TTSEVT_DATA	PCM data has been produced
TTSEVT_TAG	A text-embedded control tag has been encountered
TTSEVT_BOOKMARK	A text-embedded bookmark has been encountered
TTSEVT_ENDOFSPEECH	Audio rendering has finished
TTSEVT_NOTSENT	Audio destination internal buffers are full
TTSEVT_FREESPACE	There is some space in the audio destination internal buffers
TTSEVT_ERROR	An asynchronous error occurred
TTSEVT_PAUSE	Audio destination has been paused
TTSEVT_RESUME	Audio destination has been resumed (after Pause)
TTSEVT_SENTENCE	A new sentence has started
TTSEVT_AUDIO	An audio mixer command has started
TTSEVT_VOICECHANGE	A voice change has been encountered
TTSEVT_LANGUAGECHANGE	A language change put in the text is starting now

For each event the contents of "lData" is different and sometimes is empty. Here is a list:

Event	lData
TTSEVT_TEXT	Contains a chunk of text. lData should be casted to (char *)
TTSEVT_WORDTRANSCRIPTION	Contains a phonetically transcribed word. lData should be casted to (char *)
TTSEVT_PHONEMES	Contains a phonetic handle to be passed to ttsPhoneticUtils(TTS_GETPHONETICSTREAM…)
TTSEVT_AUDIOSTART	void
TTSEVT_DATA	Contains audio data. Must be casted to (ttsPhonInfoType *) – see 17.2
TTSEVT_TAG	Contains a tag string. Must be casted to (char *)
TTSEVT_BOOKMARK	Contains a bookmark string. Must be casted to (char *)
TTSEVT_ENDOFSPEECH	void
TTSEVT_NOTSENT	void
TTSEVT_FREESPACE	void
TTSEVT_ERROR	Contains an error message string. Must be casted to (char *)
TTSEVT_PAUSE	void
TTSEVT_RESUME	void
TTSEVT_SENTENCE	void
TTSEVT_AUDIO	Contains an audio string (that is an audio filename).
TTSEVT_VOICECHANGE	Contains a voice name.
TTSEVT_LANGUAGECHANGE	Contains a language name.

[12] Some other undocumented events are declared in "loqtts.h". They may be subjected to change without notice

12.1 Example

Here is an example of callback function that intercept the most part of events.

```
static void TTSCALLBACK myCallback(
      ttsEventType nReason,
      void *lData,
      void *pUser
) {
    ttsHandleType hInstance = (ttsHandleType)pUser;
    switch (nReason) {
    case TTSEVT_TEXT:
            printf("%s\n",(char *)lData);
            break;
    case TTSEVT_DATA:
      {
            char line[256];
            ttsPhonInfoType *p;
            p = (ttsPhonInfoType *) lData;
            sprintf(line,"%d (duration: %4d) - nbytes: %4ld\n",
                  p->ipacode,p->DurationMs,p->NBytes);
            printf("%s",line);
            break;
      }
    case TTSEVT_ERROR:
            fprintf(stderr,"%s\n",(char *)lData);
            break;
    case TTSEVT_AUDIOSTART:
          printf("BEGIN OF SPEECH!\n");
          break;
    case TTSEVT_PAUSE:
          printf("AUDIO HAS BEEN PAUSED!\n");
          break;
    case TTSEVT_RESUME:
          printf("AUDIO HAS BEEN RESUMED!\n");
          break;
    case TTSEVT_ENDOFSPEECH:
          printf("END OF SPEECH!\n");
          break;
    case TTSEVT_FREESPACE:
          printf("Audio destination is working!\n");
          break;
    case TTSEVT_NOTSENT:
          printf("Audio destination is full!\n");
          break;
    case TTSEVT_BOOKMARK:
          printf("Bookmark = %s\n",(char *)lData);
          break;
    case TTSEVT_PHONEMES:
          {
            unsigned int pos = 0, ph, dur;
            while((ttsBoolType)ttsPhoneticUtils(
                  TTS_GETPHONETICSTREAM,lData,&ph,&dur,&pos))
            {
             printf("Phoneme n. %u - duration %d - position %d\n",
             ph,dur,pos);
            }
          }
          break;
    case TTSEVT_TAG:
            printf("control tag = %s\n",(char *)lData);
            break;
    default:break;
    }
}
```

This callback must be registered in this way:
`ttsRegisterCallback(hInstance,myCallback,(void *)hInstance,0);`

Here the hInstance handle is passed to the callback function via the pUser parameter of ttsRegisterCallback.

In the same way, if you use the C++ API set, you must call the RegisterCallback method of class CttsInstance:

```
MyInstance.RegisterCallback(myCallback,myUserData,0);
```

Note that the myCallback function is a "C" function and therefore a "non-static" class member function won't work. How to access member variables from within the callback then? A possible solution is to register an "auxliary" static member function as callback, and then call the "real" callback via a class pointer. Again the pUser parameter is well suited for this purpose, as explained in the following example:

```
File "MYCLASS.H"
class MyClass {
  public:
      int Init();
            ...
  private:

      CttsInstance m_tts;

      void Callback(ttsEventType nReason,void *pData);
      static void TTSCALLBACK _Callback(ttsEventType nReason,void
                  *pData,void *pUser);
            ...
};

File "MYCLASS.CPP"
int MyClass::Init()
{
      ...
      m_tts.RegisterCallback(_Callback,this);
}

void MyClass::Callback(ttsEventType nReason,void *pData)
{
      //"real" class Callback function
}

static void TTSCALLBACK MyClass::_Callback
      (ttsEventType nReason,void *pData,void *pUser)
{
      //auxiliary callback function that reflects towards the
      //"real" function
      ((MyClass *)pUser)->Callback(nReason,void *pData);
}
```

13 Session and Instance Configuration parameters

13.1 Session parameters

You can configure a session by assigning an appropriate value to parameters and storing them into a IniFile, in the form of a “key = value” line like:

“DataPath” = “/usr/Loquendo/ LTTS/Data”

or registry section (Windows only) like:

REGEDIT4

[HKEY_LOCAL_MACHINE\Software\ Loquendo\LTTS\default.session]

"DataPath"=“c:\\Program Files\\Loquendo\\LTTS\\Data”

The full path name of the IniFile can be passed to ttsNewSession as parameter (see Reference Guide). If the IniFile name is surrounded by square brackets (“[ ]”), session parameters are searched in the registry. For instance, if you specify “[Steve.session]”, the registry section:

[HKEY_LOCAL_MACHINE\Software\ Loquendo\LTTS\ Steve.session]

is searched.

Here is the complete list of session parameters:

- **DataPath** - the only one required: defines the installation path of Loquendo TTS voices (you need to change it only if you plan to install more than one Loquendo TTS SDK on the same machine)
- **LibraryPath** – defines the installation path of the language DLLs or Shared Objects (by default it has the same value as “DataPath”)
- **PrerecordedPath** – defines the default path of RAW signal files (see RAW signal files playing in the Loquendo TTS user’s guide); by default has the same value as “DataPath”
- **LogFile** – log file full path name. Some special values are possible: “stderr” (redirects logging to console, if any), “MessageBox”[13] (pops up a window), “nul:”[14] or “/dev/null”[15] (no logging)
- **TraceFile** – full file name to trace out Loquendo TTS run-time messages. Special value: “stderr” (same as above)
- **LicenseFile** – defines the full path name of a License file (you need to set it only if you want to change the default one, installed automatically with your Loquendo TTS license)
- **LoadingMode** – Setting this parameter to “RAM” will force Loquendo TTS to load all its data in memory at startup, instead of accessing it run-time, with the purpose of speeding up TTS conversions. This setting will sensibly slow down sessions and voices initializations, and will increase a lot the amount of requested memory; for this reason it is not recommended, unless your application uses just one voice serving a large number of channels. If your application experiments difficulties in accessing voice specific files (e.g. *.bin files, even if they are present on the disk), you may set LoadingMode to “disk” instead. This setting (default) fixes some

[13] Windows ONLY
[14] Windows
[15] Linux

Windows 2000 and NT issues on memory mapped files. To force use of memory mapped files set this parameter to "map".

- **FailOnLicenseError** – By default, whenever a license error occurs, Loquendo TTS works as usual, but, instead of the requested text, an appropriate license error message is synthesized, explaining what is going wrong. This behavior is well suited for multimedia applications, but can cause troubles to telephony applications (final customers aren't interested in licensing problems). If you set this parameter to "true", however, license errors occur at Loquendo TTS initialization stage.
- **LoadLanguageGuesser** – By default, if the Mixed Language Support has been installed, the Language Guesser library is loaded at startup. Set this parameter to "false" if you want to disable it
- **DisableGlobalQuery** – By default, a full query of all installed voices is carried out at startup. Set this parameter to "true" if you want to disable it. This will speed up session opening, but will slow down voice switching.
- **OldProsodyRange** – By default, the range for prosody parameters (pitch, speed and volume) is 0 - 100. Set this parameter to "true" if you want to restore the old range (0 –10), which has been used up to the 6.3.x versions.
- **UrlProxy** – When using URL address for playing files or loading lexicons, a proxy can be set with this key (this is an instance parameter too).
- **UrlUsername** – When using URL address for playing files or loading lexicons, the username can be set with this key (this is an instance parameter too).
- **UrlPassword** – When using URL address for playing files or loading lexicons, the password can be set with this key (this is an instance parameter too).
- **UrlTimeOut** – When using URL address for playing files or loading lexicons, the time out (in msec) can be set with this key (this is an instance parameter too).
- **UrlPort** – When using URL address for playing files or loading lexicons, the port number (es. 8080) can be set with this key (this is an instance parameter too).

13.2 Instance parameters

Like sessions, you can configure your instances by assigning an appropriate value to parameters and storing them into a IniFile, in the form of a "key = value" line like:

"InputTextCoding" = "unicode"

or registry section (Windows only) like:

REGEDIT4

[HKEY_LOCAL_MACHINE\Software\Loquendo\LTTS\MyApplication]

"InputTextCoding"="unicode"

The full path name of the IniFile can be passed to ttsNewInstance as parameter (see Reference Guide). Again, If the IniFile name is surrounded by square brackets ("[ ]"), instances parameters are searched in the registry. For instance, if you specify "[MyApplication]", the registry section:

[HKEY_LOCAL_MACHINE\Software\Loquendo\LTTS\ MyApplication]

is searched.

Here is the complete list of instance parameters:

- **OEM** – If you set this parameter to true, input text is considered OEM coded (Windows only) instead of ANSI
- **MultiCRPause** – Usually empty lines in text generate a pause. If you set this parameter to "FALSE", no pause is generated.
- **MultiSpacePause –** Usually multiple spaces or tabs in text generate a pause. If you set this parameter to "FALSE", no pause is generated.
- **MaxParPause** – Usually lines short than 5 words (like titles or signatures) are automatically terminated by a pause. You can change this value from 5 to a different value; use "0" (zero) if you want to disable this feature
- **SpellingLevel** – Three possible values: "normal", "spelling" (all words are spelled out), "pronounce" (no words are spelled out)
- **TaggedText** – If you set this parameter to "false", control tags are not processed but pronounced.
- **TraceWordTranscription** – If you set this parameter to "true", phonetic transcription strings are traced in the TraceFile (if enabled)
- **InputType** – Sets up the TTSDEFAULT value for the "InputType" parameter of ttsRead. Possible values: "file" or "buffer". If no InputType has been set, the value for TTSDEFAULT is "buffer".
- **InputTextCoding** - Sets up the TTSDEFAULT value for the "TextCoding" parameter of ttsRead. Possible values: "unicode", "utf-8", "iso", "ansi" (default) If no InputTextCoding has been set, the value for TTSDEFAULT is "ansi".
- **ReadingMode** - Sets up the TTSDEFAULT value for the "ReadingMode" parameter of ttsRead. Possible values: "ssml", "paragraph", "autodetect", "multiline" . If no ReadingMode has been set, the value for TTSDEFAULT is "multiline".
- **ProcessingMode** - Sets up the TTSDEFAULT value for the "ProcessingMode" parameter of ttsRead. Possible values: "blocking", "slice", "nonblocking". If no ProcessingMode has been set, the value for TTSDEFAULT is "nonblocking".
- **LogFile** – You can override the value specified in the session IniFile
- **TraceFile** - You can override the value specified in the session IniFile
- **AutoGuess –** Activates and configures the AutoGuess mode (See Mixed Language for details). Requires the Additional CD "Mixed Language Capabilities". The syntax of this configuration parameter is:

  AutoGuess=[Type]**:**[Language list]

  Possible values for "type":

  1. **"no" –** no AutoGuess mode
  2. **"VoiceParagraph" –** Detects language and changes voice accordingly *paragraph by paragraph*
  3. **"VoiceSentence" -** Detects language and changes voice accordingly *sentence by sentence*
  4. **"VoicePhrase" -** Detects language and changes voice accordingly *phrase by phrase*
  5. **"LanguageParagraph" –** Detects and change language *paragraph by paragraph* without changing the active voice

6. **"LanguageSentence" –** Detects and change language *sentence by sentence* without changing the active voice
7. **"LanguagePhrase" –** Detects and change language *phrase by phrase* without changing the active voice
8. **"LanguageWord" –** Detects and change language *word by word* without changing the active voice
9. **"BothParagraphSentence" –** Combines the effects of "**VoiceParagraph"** and "**LanguageSentence"**
10. **"BothParagraphPhrase" –** Combines the effects of "**VoiceParagraph"** and "**LanguagePhrase"**
11. **"BothParagraphWord" –** Combines the effects of "**VoiceParagraph"** and "**LanguageWord"**
12. **"BothSentencePhrase" –** Combines the effects of "**VoiceSentence"** and "**LanguagePhrase"**
13. **"BothSentenceWord" –** Combines the effects of "**VoiceSentence"** and "**LanguageWord"**
14. **"BothPhraseWord" –** Combines the effects of "**VoicePhrase"** and "**LanguageWord"**

The AutoGuess keyword requires a comma-separated language list (e.g. English, French, Spanish, German). For types 9-14 a postponed '-' (minus) character (e.g. "**Swedish-**") means that voice changes are admitted, but not "language only" changes (see the second example below).

A prefixed '-' (minus) means that only language changes are admitted (not voice changes).

Some examples:

`AutoGuess=VoiceSentence:Italian,English` (sentence by sentence changes among Italian and English voices)

`AutoGuess=BothSentenceWord:French-,Spanish-,English` (sentence by sentence detects the right language and changes voice accordingly. In addition, while speaking with non-English voices, English words are detected and pronounced with the English phonetic rule set).

- **LanguageSetForGuesser** – Defines the comma-separated list of languages the Language Guesser should guess among (es: "English, French, Spanish, Italian") – Requires the Additional CD "Mixed Language Capabilities"
- **DefaultNumberType –** Defines the default number type among: "generic" (default), "telephone", "currency", "code", "hour", "date", "MasculineOrdinal", "FeminineOrdinal"
- **SampaSecondAccent** – If you set this parameter to "NO", SAMPA secondary stress (the '%' character) is simply skipped. By default, the SAMPA secondary stress is converted to the SAMPA primary stress (the '"' character).

N.B. Every Instance parameter can be changed also with an appropriate tag embedded in the input text: **\@key=value**. For instance, the following tag:

```
\@AutoGuess=VoiceSentence:Italian,French
```

activates the automatic voice switching, starting from next sentence.

The tag `\@DefaultNumberType=telephone` changes the default number pronunciation, from “generic” to “telephone”.

Instance parameters that has been changed with the `\@` tag keep the new value, until a new tag is encountered or the instance is closed.

14 Migration from Actor 5.x

Migrating an application written for Loquendo TTS 5.x to Loquendo TTS 6.x is quite simple.

- First of all you need to open a session (using ttsNewSession) since sessions were not present in LTTS 5. You don't need to close channels anymore, because instances are automatically closed when you close your session.
- Second, you have to remove any occurrence of LTTS 5.x audio destination calls, that are now implemented by an external library (you just need to call a single API: ttsSetAudio).
- Third, declare ttsHandleType any session, instance, voice and lexicon handles (previously channel handles were declared UINT, while voices and lexicon handles didn't exist).

Here is a basic replacement table, including the most important constants, types and function names:

OLD NAME	TYPE	NEW NAME
#include "actor.h"	header file	#include "loqtts.h"
#include "wavdest.h"	header file	*remove*
#include "fildest.h"	header file	*remove*
#include "filewav.h"	header file	*remove*
WavInit, FilInit, FWInit, WavClose, FilClose, FWClose	audio destinations calls	*remove*
ttsErrType	typedef	ttsResultType
ttsOpen	LTTS call	ttsNewInstance and ttsSetAudio (optional)
ttsClose	LTTS call	ttsDeleteInstance (optional)
ttsSetVoice	LTTS call	ttsNewVoice or ttsActivateVoice
ttsLoadVoice	LTTS call	ttsNewVoice
ttsUnloadVoice	LTTS call	ttsDeleteVoice (optional)
ttsUnloadVoices	LTTS call	*remove*
ttsSwitchVoice	LTTS call	ttsActivateVoice
ttsGetErrorMessage	LTTS call	ttsGetError
ttsSetAudioDest	LTTS call	ttsSetAudio
ttsSetAudio	LTTS call	*remove*
ttsSetLexicon	LTTS call	ttsNewLexicon
ttsReadText	LTTS call	ttsRead
ttsReadTextFile	LTTS call	ttsRead
ttsSetText	LTTS call	ttsRead
ttsReadTextSlice	LTTS call	ttsRead
ttsWaitEnd	LTTS call	*remove (use ReadText's*

		TTSBLOCKING mode)
ttsSetReadingMode	LTTS call	*remove (use ReadText's input modes)*
ttsGetIniVal	LTTS call	ttsLoadConfigurationParam
ttsSetIniVal	LTTS call	ttsSaveConfigurationParam
TtsEnum	LTTS call	ttsQuery

15 Microsoft SAPI 5 support

15.1 Getting started

This section contains some notes that may help working with Loquendo SAPI 5 interfaces. For a description of SAPI 5 interfaces, you may download the appropriate documentation by Microsoft from:

http://download.microsoft.com/download/speechSDK/SDK/5.1/WXP/EN-US/sapi.chm

15.2 Known Limitations and bugs

This version of Loquendo TTS SAPI5 interface library has some limitations. Although the engine passes the Microsoft SAPI 5 compliance test, some interfaces are not implemented in this version. The limitations are described below.

15.2.1 Language identification

Some engines (especially non-English ones) may return language identifiers (such as Chilean or Catalan) that are not recognized by the Control Panel Speech Applet. **This is a known limitation of the SAPI5 specification that cannot be removed within the present Microsoft SAPI release.**

15.2.2 Non linear coding support

If only non-linear coding voices are installed, i.e. only A-law or µ-law, the Control Panel Speech Applet may not work correctly with Loquendo voices; in addition the Microsoft SAPI5 shipped application "TTSApp" may not speak unless you change the selected item in the Audio format combo box. Loquendo TTS is not responsible for this bad behavior, which seems to be due to the SAPI5 Multimedia Audio destination.

Therefore, in order to correctly use the SAPI interface with Loquendo TTS, please don't forget to always install Loquendo linear coding voices.

15.2.3 ISpTTSEngine

The attributes SPF_SPEAK_NLP_PUNC and SPF_PURGEBEFORESPEAK attributes of **Speak** method are not supported.

15.2.4 Lexicon

Lexicon handling is supported for all Loquendo TTS engines.

15.2.5 Phoneme

The SAPI5 limitations described above make it difficult to deal with phonetic alphabets that do not consider other than english/chinese/japanese phonemes. In order to let user manage valid phonemes even with non-English voices, a specific configuration has been provided that make it possible to use phoneme information even with Greek, Italian and Brazilian voices.

In order to enable this, you should enter the following string in the registry:

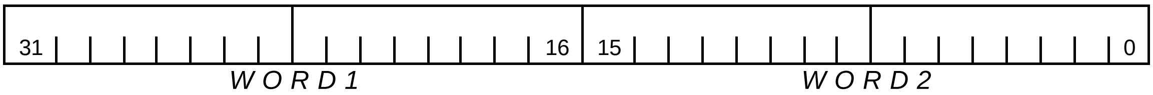
```
REGEDIT4
[HKEY_LOCAL_MACHINE\SOFTWARE\loquendo\TTS\LoqSapi5]

"LoquendoPhonCode" = "TRUE"
```

In this configuration, a 32bits value is generated for each Phoneme event. The MSB (bits 16 to 31), are sent in the PhoneID field of current phoneme, the LSB (bits 0 to 15) are sent in the PhoneID field of next phoneme.

More precisely:

LoquendoPhonCode	
.PhoneID	.NextPhoneID



The obtained 32 bits value identifies a “Loquendo TTS” phoneme (see the phoneme table in the Appendix A).

For a full description of the Loquendo TTS Phonemes, see the Loquendo TTS User Guide.

If the configuration above is not enabled, all engines (even non-English ones!) return English PhoneIDs (as described in the SAPI5 manual). The mapping non-English phonemes / English PhoneIDs although as accurate as possible, may not be precise.

15.2.6 Viseme

In both the above configurations Visemes are fully supported.

15.2.7 XML SAPI tags limitations

XML <partofsp> - part of speech tag is not implemented.

In the <context> tag, when the ID attribute is set to “time” the engine do not generate minutes/seconds normalizations, as in the following example.

```
<context ID = "time">1'21"</context>
is normalized as "one minute twenty one seconds"
or "one minute and twenty one seconds"
```

In the <context> tag, when the ID attribute is set to “address” the engine:

- do not generate the US state name from the state code;
- do not spell the street number.

Loading the SAPI5TAGS.XML file, supplied by Loquendo, in the TTSApp application, may test all SAPI5 tags. The file is very close to the Appendix B content.

It’s important to note that Microsoft has overridden the XML specifications in two different aspects:

- Tags are case-insensitive;
- The phoneme set uses the “**&”** character that normally cannot be used as-is in an XML document and it must be replaced by the escape sequence **&amp**.

Due to this peculiarity Microsoft Internet Explorer cannot read a SAPI 5 fully compliant file.

15.2.8 Loquendo TTS Phoneme table

Loquendo TTSPhoneme	Loquendo PhonCode	Loquendo TTSPhoneme	Loquendo PhonCode
#	0x34C00000	$:	0xA8A20000
$	0xA8C00000	@	0xEEC00000

Loquendo TTSPhoneme	Loquendo PhonCode
AR	0x4CC00000
Aa:	0x4EA20000
Ae	0x36C00000
Ah	0xB4C00000
Ao	0x50C00000
Ao)	0x50C60000
Bh	0xE8C00000
Dg	0xDEC00000
Dg:	0xDEA20000
Dh	0x3AC00000
Dz	0xDCC00000
Dz:	0xDCA20000
E	0x62C00000
E)	0x62C60000
E):	0x62A23000
E:	0x62A20000
GL	0x16A60000
GN	0x1AA60000
Gh	0x70C00000
Gl	0xB8C00000
Gl:	0xB8A20000
Gn	0x8CC00000
Gn:	0x8CA20000
HAI	0x01BE0000
HAU	0x01D80000
HB	0x02AB0000
HD	0x06AB0000
HEE	0x09AF0000
HEI	0x09BE0000
HEU	0x5FD80000
HEh	0x5EC00000
HG	0x0CAB0000
HH	0x78C00000
HIE	0x7DAF0000
HOI	0x55BE0000
HOU	0x1DD80000
HR	0x98B60000
HRB	0x98545800
HRF	0xA4C00000
HTh	0x26AC0000
HUE	0xB1AF0000
Hd	0x3AAB0000
Hg	0xC4AC0000
Hh	0x76C00000
Hj	0x12C00000
Hk	0xC4C00000
Hl	0x16B60000
Hn	0x1AB60000
Hr	0x98C00000
Hub	0x02CE0000
Hud	0x06CE0000
Hug	0x0CCE0000
Huk	0x14CE0000
Hup	0x1ECE0000
Hut	0x26CE0000
Hv	0x2AAB0000
Hw	0x2CC00000
Hy	0x74C00000
Hz	0x32AB0000
I	0x7CC00000
J	0x6AC00000
L	0x16C00000
M	0x8AC00000
N	0x40C00000
OE	0x3CC00000
OE:	0x3CA20000
OR:	0x64A20000
Oa	0x54C00000

Loquendo TTSPhoneme	Loquendo PhonCode
Oa:	0x54A20000
Oe	0x42C00000
Oe)	0x42C60000
Ou	0xBAC00000
Pf	0x1F850000
R	0xA4C00000
RD	0x58C00000
RL	0x82C00000
RN	0x8EC00000
RT	0xACC00000
Rs	0xA6C00000
Rz	0xACC00000
T$	0xE4C00000
T$:	0xE4A20000
Th	0xEAC00000
Ts	0xE2C00000
Ts:	0xE2A20000
U	0xB0C00000
UO	0xAEC00000
Y	0xD0C00000
Zg	0xC0C00000
Zh	0xEAAC0000
`Aa:	0x4E401000
`Aa:1	0x4E401700
`Aa:2	0x4E401780
`Ae	0x36A00000
`Ae1	0x36A0E000
`Ae2	0x36A0F000
`Ae:1	0x36A2E000
`Ae:2	0x36A2F000
`Ah	0xB4A00000
`Ao	0x50A00000
`Ao)	0x50A03000
`E	0x62A00000
`E)	0x62A03000
`E):	0x62401180
`E1	0x62A0E000
`E2	0x62A0F000
`E:	0x62401000
`E:1	0x62401700
`E:2	0x62401780
`HAI	0x016F8000
`HAU	0x01760000
`HEE	0x096BC000
`HEI	0x096F8000
`HEU	0x5F760000
`HIE	0x7D6BC000
`HOI	0x556F8000
`HOU	0x1D760000
`HUE	0xB16BC000
`I	0x7CA00000
`I1	0x7CA0E000
`I2	0x7CA0F000
`OE	0x3CA00000
`OE1	0x3CA0E000
`OE2	0x3CA0F000
`OE:	0x3C401000
`OE:1	0x3C401700
`OE:2	0x3C401780
`OR:	0x64401000
`Oa	0x54A00000
`Oa1	0x54A0E000
`Oa2	0x54A0F000
`Oa:	0x54401000
`Oe	0x42A00000
`Oe)	0x42A03000
`Oe1	0x42A0E000
`Oe2	0x42A0F000

Loquendo TTSPhoneme	Loquendo PhonCode
`Oe:1	0x42A2E000
`Oe:2	0x42A2F000
`Ou	0x30A00000
`Ou1	0xBAA0E000
`Ou2	0xBAA0F000
`U	0xB0A00000
`U1	0xB0A0E000
`U2	0xB0A0F000
`UO1	0xAE401700
`UO2	0xAE401780
`UO:1	0xAE401700
`UO:2	0xAE401780
`a	0x00A00000
`a)	0x00A03000
`a):	0x00C60000
`a1	0x00A0E000
`a2	0x00A0F000
`a:	0x00401000
`e	0x08A00000
`e)	0x08A03000
`e:	0x08401000
`e:1	0x08401700
`e:2	0x08401780
`i	0x10A00000
`i)	0x10A03000
`i:	0x10401000
`i:1	0x10401700
`i:2	0x10401780
`o	0x1CA00000
`o)	0x1CA03000
`o):	0x1C401180
`o:	0x1C401000
`o:1	0x1C401700
`o:2	0x1C401780
`u	0x28A00000
`u)	0x28A03000
`u:	0x28401000
`u:1	0x28401700
`u:2	0x28401780
`y	0x30A00000
`y:	0x30401000
`y:1	0x30401700
`y:2	0x30401780
a	0x00C00000
a)	0x00C60000
a):	0x00A23000
a:	0x00A20000
a~	0x00B70000
b	0x02C00000
b:	0x02A20000
c	0x04AF0000
ch	0x56C00000
d	0x06C00000
d:	0x06A20000
e	0x08C00000
e)	0x08C60000
e:	0x08A20000
e~	0x08B70000
f	0x0AC00000
f:	0x0AA20000
g	0x0CC00000
g:	0x0CA20000
h	0x0EC00000
i	0x10C00000
i)	0x10C60000
i:	0x10A20000
i~	0x10B70000
j	0x12C00000

Loquendo TTSPhoneme	Loquendo PhonCode
k	0x14C00000
k:	0x14A20000
l	0x16C00000
l:	0x16A20000
m	0x18C00000
m:	0x18A20000
n	0x1AC00000
n:	0x1AA20000
o	0x1CC00000
o)	0x1CC60000
o):	0x1CA23000
o:	0x1CA20000
o~	0x1CB70000
p	0x1EC00000
p:	0x1EA20000
r	0xA0C00000
s	0x24C00000
s:	0x24A20000
t	0x26C00000
t:	0x26A20000
u	0x28C00000
u)	0x28C60000
u:	0x28A20000
u~	0x28B70000
v	0x2AC00000
v:	0x2AA20000
w	0x2CC00000
x	0x2EC00000
y	0x30C00000
y:	0x30A20000
z	0x32C00000

15.2.9 Loquendo XML SAPI tags

These tags may be tested with TTSApp, by loading the file `SAPI5TAGS.XML` supplied by Loquendo.

<u>VOICE STATE CONTROL TAGS.</u>

VOLUME	<volume level="100"> This text should be spoken at the default volume. <volume level="50"> This text should be spoken at volume level fifty. </volume> </volume> <volume level="80"/> All text which follows should be spoken at volume level eighty. <volume level="100"/> This text should be spoken at the default volume.
RATE	<rate absspeed="0"/> All text which follows should be spoken at the default rate. <rate absspeed="5"> This text should be spoken at rate five. <rate absspeed="-5"> This text should be spoken at rate negative five. </rate> </rate> <rate absspeed="10"/> All text which follows should be spoken at rate ten. <rate absspeed="0"/> All text which follows should be spoken at the default rate.
SPEED	<rate speed="5"> This text should be spoken at rate five. <rate speed="-5"> This text should be spoken at rate zero. </rate> </rate>
PITCH	<pitch absmiddle="0"/> All text which follows should be spoken at the default pitch. <pitch absmiddle="5"> This text should be spoken at pitch five. <pitch absmiddle="-5"> This text should be spoken at pitch negative five. </pitch> </pitch> <pitch absmiddle="10"/> All text which follows should be spoken at pitch ten. <pitch absmiddle="0"/> All text which follows should be spoken at the default pitch.
EMPH	Next sentence should emphasize the words "said" and "away". I <emph>said</emph>, "Go <emph>away</emph>". Next sentence instead do not emphasizes them. I said, "Go away".
SPELL	<spell> These words should be spelled out. </spell> These words should not be spelled out.

DIRECT ITEM INSERTION TAGS.

SILENCE	Five hundred milliseconds of silence <silence msec="500"/> just occurred.
PRON	<pron sym="h eh 1 l ow & w er 1 l d "/>. <pron sym="h eh 1 l ow & w er 1 l d"> hello world </pron>.
BOOKMARK	A bookmark event <bookmark mark="bookmark_one"/> should just occurred.

VOICE SELECTION TAGS.

VOICE	<voice required="name=Dave"> Hi! My name is Dave. My voice is generated by the Loquendo text-to-speech synthesis system. </voice> <voice required="name=Susan"> Hi! My name is Susan. My voice is generated by the Loquendo text-to-speech synthesis system. </voice>
LANG	<lang langid="409"> A US English voice should speak this. </lang>
CUSTOM PRONUNCIATION	<P DISP="disp" PRON="pron">word</P>.

ENGLISH VOICE CONTEXT CONTROL TAGS.

PARTOFSP	NOT SUPPORTED

CONTEXT - DATE:

ID=date_mdy	<context ID = "date_mdy">12/21/99</context> will be normalized to "December twenty first ninety nine" or "December twenty first nineteen ninety nine". <context ID = "date_mdy">12/21/1999</context> will be normalized to "December twenty first nineteen ninety nine".
ID=date_dmy	<context ID ="date_dmy">21.12.99</context> will be normalized to "December twenty first ninety nine" or "December twenty first nineteen ninety nine". <context ID = "date_dmy">21-12-1999</context> will be normalized to "December twenty first nineteen ninety nine".
ID=date_ymd	<context ID = "date_ymd">99-12-21</context> will be normalized to "December twenty first ninety nine" or "December twenty first nineteen ninety nine". <context ID = "date_ymd">1999.12.21</context> will be normalized to "December twenty first nineteen ninety nine".
ID=date_ym	<context ID = "date_ym">99-12</context>

	will be normalized to "December ninety nine" or "December nineteen ninety nine". <context ID = "date_ym">1999.12</context> will be normalized to "December nineteen ninety nine".
ID=date_my	<context ID = "date_my">12/99</context> will be normalized to "December ninety nine" or "December nineteen ninety nine". <context ID = "date_my">12/1999</context> will be normalized to "December nineteen ninety nine".
ID=date_dm	<context ID = "date_dm">21.12</context> will be normalized to "December twenty first".
ID=date_md	<context ID = "date_md">12/21</context> will be normalized to "December twenty first"
ID=date_year	<context ID = "date_year">1999</context> will be normalized to "nineteen ninety nine". <context ID = "date_year">2001</context> will be normalized to "Two thousand one".

CONTEXT - TIME:

ID=time	<context ID = "time">12:30</context> will be normalized to "twelve thirty". <context ID = "time">01:21</context> is normalized as "one twenty one" or "oh one twenty one".

CONTEXT - NUMBER:

ID=number_cardinal	<context ID = "number_cardinal">3432</context> will be normalized to "three thousand four hundred and thirty two".
ID=number_digit	<context ID = "number_digit">3432</context> will be normalized to "three four three two".
ID=number_fraction	<context ID = "number_fraction">3/15</context> will be normalized to "three fifteenths" or "three over fifteen".
ID=number_decimal	<context ID = "number_decimal">423.1243</context> will be normalized to "four hundred and twenty three point one two four three".
ID=phone_number	The exact implementation may be defined in a future release of SAPI. In the current Loquendo implementation the number <context ID = "phone_number">0117576207</context> will be normalized to "oh eleven seven five seven six two oh seven".
ID=currency	<context ID = "currency">$34.90</context> will be normalized to "thirty four dollars and ninety cents".

CONTEXT - WEB (web_url):

ID=web_url	<context ID = "web_url">www.Microsoft.com</context> will be normalized to "w w w dot Microsoft dot com".

CONTEXT - E-MAIL (e-mail address):

`ID=e-mail_address`	`<context ID = "e-mail_address">someone@microsoft.com</context>`
	`is normalized to "Someone at Microsoft dot com".`

CONTEXT - ADDRESS:

`ID=address`	`<context ID = "address">One Microsoft Way, Redmond, WA, 98052</context>`
	`will be normalized to "One Microsoft Way Redmond Washington nine eight zero five two".`

CONTEXT - ADDRESS POSTAL:

`ID=address_postal`	`<context ID = "address_postal">A2C 4X5</context>`
	`will be normalized to "A 2 C 4 X 5".`

15.3 Lexicon files

For each supported language, you can specify an additional lexicon file to be used with the SAPI 5 interfaces.

You can specify these files inserting the following registry string values:

HKEY_LOCAL_MACHINE\SOFTWARE\Loquendo\LTTS\LoqSapi5\Lex.<LangId>=<lexicon_file_path>

where <LangId> is a decimal integer specifying the Microsoft Windows language identifier (e.g. 1033 for US english; see Microsoft documentation for a complete list of language identifiers).

16 Microsoft SAPI 4 support

16.1 Getting started

This section contains some notes that may help working with Loquendo SAPI 4 interfaces. For a description of SAPI 4 interfaces, you may download the appropriate Microsoft documentation from:

http://www.microsoft.com/speech/download/old/sdk40a.asp

16.2 Limitations

The current version of the Loquendo TTS engine does not support some optional SAPI 4 features:

- the ITTSAttributes::RealTimeGet() and ITTSAttributes::RealTimeSet() methods (Engine object);
- the CHARSET_ENGINEPHONETIC and CHARSET_IPAPHONETIC character sets for the ITTSCentral ::TextData() method (Engine object);
- the ILexPronounce and ILexPronounce2 interfaces (Engine object);
- the ITTSBufNotifySink::WordPosition() notification (Engine object);
- the ITTSNotifySink::Visual() notification (Engine object);
- the ITTSNotifySink2 notification interface (Engine object).

Finally, note that the ITTSCentral::Inject() method (Engine object) takes effect only on the next paragraph of the text being spoken.

16.3 Audio destinations

Since the Loquendo TTS engine has a different behavior according to if the audio destination is real time (e.g. an audio board) or not (e.g. a file), the following assumption is made on the Audio object passed to the engine (through the ITTSEnum::Select() method, for instance):

- if the Audio object supports the IAudioMultiMediaDevice, the IAudioDirect or the IAudioTel interface, then it is assumed to be a real time destination;
- otherwise, if the Audio object supports the IAudioFile interface, then it is assumed to be a non-real time destination;
- otherwise, the Audio object is assumed to be a real time destination again.

16.4 Lexicon files

For each supported language, you can specify an additional lexicon file to be used with the SAPI 4 interfaces.

You can specify these files through the *Loquendo TTS SAPI4 lexicon files* dialog box that can be invoked through the ITTSDialogs::LexiconDlg() method.

17 Audio destination

Loquendo TTS, like any other text-to-speech system, is basically a piece of software that receives a written text as input and produces a synthetic waveform that sounds like a nearly human speech of that text. However, the output destination of the speech conversion is application dependent: for instance, you may want to produce a file, or drive the output to the multimedia board, or to a telephony card. Since it is virtually impossible to know in advance where the output has to be sent, the so called "Audio destination" has been designed as a separate library (more precisely a DLL or Shared Object) that can be loaded when requested.

Three built-in audio destinations are shipped with Loquendo TTS, the "LoqAudioBoard" library, which outputs to the multimedia board (both for Linux and Windows), the "LoqAudioFile" library, which produces an audio file (Raw signal file or .WAV file - as well available for both Windows and Linux) and the "LoqAudioAsf" library which produces audio files compliant with the Windows Advanced File Streaming format (compressed streaming file format using the embedded audio codecs installed with Windows Media Player, version 9 required).

To use one of those Audio destinations, simply pass the name of the chosen library to ttsSetAudio API. More precisely, to use the multimedia audio destination you have to call:

```
ttsSetAudio(hInstance,”LoqAudioBoard”, NULL, “L”, 0);
```

Note that the third parameter is the name of the audio device. NULL is the default one, corresponding to WAVE_MAPPER (Windows) and /dev/audio (Linux). You can pass a different string (for instance "1", if you want to use device # 1 if you have more than one audio board in your Windows system).

The fourth parameter is the speech coding (in the example above, the "linear" audio coding has been specified). The fifth parameter is reserved for future scope.

To use the audio file destination you can call:

```
ttsSetAudio(hInstance,”LoqAudioFile”, “myfile.wav”, “L”, 0);
```

Here the third parameter is the name of the output file. A file name with the extension ".wav" means that you want to produce a .WAV audio file; any other extension means that you want a RAW audio file (with no header). The special character '?' can be used to specify that you want output to a different audio file for each ttsRead call.

For instance:

```
ttsSetAudio(hInstance,”LoqAudioFile”, “myfile?.pcm”, “L”, 0);
ttsRead(hInstance,”first sentence”,……);
ttsRead(hInstance,”second sentence”,……);
```

produces "myfile0.pcm" containing "first sentence" and "myfile1.pcm" containing "second sentence".

You can put a '?' character anywhere in the file name, but any other '?' after the first one is ignored.

Instead, if the '?' is absent, the output is appended to the same file. Note that the audio file is opened at the ttsSetAudio call and remains opened until a ttsSetAudio call is given **with a different filename as parameter** (or ttsDeleteInstance is called). For this reason, if you need to use always the same file name for each ttsRead call, simply call

```
ttsSetAudio(hInstamce,NULL,NULL,NULL,0);
```

before ttsRead, as in the following example:

```
ttsSetAudio(hInstance,”LoqAudioFile”, “myfile.wav”, “L”, 0);\
/* ttsRead keeps control until the end */..
ttsRead(hInstance, "This is a sentence",  TTSBUFFER, TTSANSI, TTSDEFAULT,
TTSBLOCKING);
/* now use myfile.wav - then explicitly close the audio file */
ttsSetAudio(hInstamce,NULL,NULL,NULL,0);
```

17.1 Implementing your own audio destination

The source code of LoqAudioFile is included in the Loquendo TTS SDK, so in principle you could develop your own custom audio destination, modifying LoqAudioFile according to your needs and recompiling it. A real empty-stub for developing an audio destination is LoqAudioStub (on the LTTS SDK too).

Basically Loquendo TTS and the audio destination communicate by means of 6 callback functions. A table of function pointers, (here declared in the LoqAudioFile function) is initialized with up to 8 callback function names. Since none of those functions is strictly required, the number of rows can be smaller than 8 (LoqAudioFile has only 6 callback functions); the assignment depends on the first column, where the appropriate mnemonic (declared in "loqtts.h") is listed:

```
{tts_GET_VERSION, (ttspFuncType)LoquendoAudioFileGetVersion},
{TTSAUDIO_OPEN, (ttspFuncType)LoquendoAudioFileOpen},
{TTSAUDIO_CLOSE, (ttspFuncType)LoquendoAudioFileClose},
{TTSAUDIO_PUTDATA, (ttspFuncType)LoquendoAudioFilePutData},
{TTSAUDIO_STOP, (ttspFuncType)LoquendoAudioFileStop},
{TTSAUDIO_GETERRORMESSAGE, (ttspFuncType)LoquendoAudioFileGetErrorMessage}
```

Here is the correspondence mnemonic/function:

tts_GET_VERSION	ttsResultType myGetVersion(char * strversion // **[OUT]**); This callback function should fill string "strversion" (max 256 characters) with a mnemonic describing the audio destination version (e.g. "My audio destination version 2.0")
TTSAUDIO_OPEN	ttsResultType myOpen(void **channel, // **[OUT]** const char *DeviceName, // **[IN]** unsigned int SampleRate, // **[IN]** const char *coding, // **[IN]** ttsBoolType *bRealTime // **[OUT]**); Loquendo TTS will call this function whenever it needs to open the audio destination. It should returns a valid audio channel handle (as first argument), eventually allocating memory for it, and a Boolean value (as fifth argument) stating whether this audio destination must be considered real-time, that is whether it plays waveforms in real time (like a board) or not (like a file). A device name can be specified, as well as the Sample rate and the audio coding
TTSAUDIO_CLOSE	ttsResultType myClose(void *channel // **[IN]**); Loquendo TTS will call this function whenever it needs to close the audio destination. It must free audio channel memory

	allocated by myOpen
TTSAUDIO_PUTDATA	ttsResultType myPutData(void *channel, // **[IN]** void *pData, // **[IN]** unsigned int nBytes, // **[IN]** unsigned int *nSent// **[OUT]**); Loquendo TTS will call this function whenever it needs to send audio samples to the audio destination. pData is a pointer to a buffer of nBytes sample. nSent should be set to nBytes, in case of success, and to 0 “zero” in case of error. This callback function is the only one that is strictly necessary (otherwise no output will be performed). Any other function is optional
TTSAUDIO_STOP	ttsResultType myStop(void *channel // **[IN]**); Loquendo TTS will call this function whenever it needs to stop (abort) audio rendering
TTSAUDIO_GETERRORMESSAGE	ttsResultType myGetErrorMessage(ttsResultType ErrorCode, // **[IN]** char *message, // **[OUT]** unsigned int size // **[IN]**); This function should fill the string “message” up to “size” characters, explaining the meaning of any possible value of “ErrorCode” (return values of your audio destination function)
TTSAUDIO_PAUSE	ttsResultType myPause(void *channel // **[IN]**); Loquendo TTS will call this function whenever it needs to stop (pause) audio rendering – only run-time audio destinations can implement this.
TTSAUDIO_RESUME	ttsResultType myResume(void *channel // **[IN]**); Loquendo TTS will call this function whenever it needs to resume audio rendering after a pause – only run-time audio destinations can implement this.

17.2 Managing audio directly from the Application

There is another possible approach to audio destination development, easier to implement as long as your audio destination is trivial (e.g. has only a PutData function). As soon as Loquendo TTS has produced a buffer of PCM samples, usually the equivalent of a phoneme, it calls your application callback with the TTSEVT_DATA event). See the Events reference for details on intercepting Loquendo TTS events on an application callback.

In your callback function you should write something like:

```
static void TTSCALLBACK myCallback(
      ttsEventType nReason,
      void *lData,
      void *pUser
) {
    switch (nReason) {
      case TTSEVT_DATA:
      {
       ttsPhonInfoType *p;
       p = (ttsPhonInfoType *) lData;
       /* now access ttsPhonInfoType fields */
```

See ttsPhonInfoType declaration in "loqtts.h":

```
typedef struct {
    void *buffer;             /* PCM buffer                     */
    unsigned long size;       /* PCM buffer size in bytes       */
    unsigned long ipacode;    /* current phoneme IPA code       */
    unsigned long NBytes;     /* current phoneme size in bytes  */
    unsigned short DurationMs;/* duration in msec               */
    unsigned char bValid;     /* current phoneme is complete    */
    char tag[ttsSTRINGMAXLEN];/* optional tag for current phoneme */
} ttsPhonInfoType;
```

"buffer" contains "NBytes" bytes of audio samples (this value will always be smaller or equal to "size"). Other information is provided, like the IPA code of current phoneme, its duration, whether the phoneme is complete or it has been split into more than a buffer and the list of control tags embedded in text before current phoneme. All this information may be useful whenever processing of visual parameters (such as "visemes") is required. This makes this approach the recommended one for implementing avatars.

18 LoquendoTTS ActiveX

NOTE: This section applies only to Loquendo TTS for Windows.

Loquendo TTS package contains an ActiveX (LoqActiveXW.ocx), exporting the main functionalities offered by the “tts” API.

An ActiveX component is an executable code (i.e. an .exe, .dll or .ocx) compliant to the Microsoft® COM® (Component Object Module or OLE® Automation) programming model. This software component can be used from an external application through its Methods and Properties that the object exhbits conforming to the Object Oriented paradigm.

Although the native C/C++ APIs (exported by the Loquendo TTS DLL) are probably the best choice for developing complex applications, since they control any aspect of Loquendo TTS in a more powerful and flexible way, application developers using high level / scripting languages such as Visual Basic, Delphi, VB script, JavaScript, etc. may find easier to interface with the ActiveX. All the methods, properties and events are described in the Table 18.1.

Pay attention: up to the previous 6.3.x versions, the ranges for “Pitch”, “Speed” and “Volume” were 0 to 10.

Three samples are included showing how to integrate LoqActiveXW.ocx in applications: LoqActiveX_VBSample (VisualBasic), HelloTTS_HTML (html + javascript) and HelloTTS_Server (ASP).

18.1 LoqActiveXW.ocx: methods, properties and events

Methods:	Description	Visual Basic sample	Notes
Init	Initialize text-to-speech engine	LoqActiveX.Init	Must be called first
Close	Close LoquendoTTS instance	LoqActiveX.Close	
Read	Performs text to speech conversion	LoqActiveX.Read "Hello World"	
Pause	Pauses the TTS	LoqActiveX.Pause	
Resume	Resume the TTS	LoqActiveX.Resume	
Stop	Stop the TTS	LoqActiveX.Stop	
Record	Save speech to a PCM file	bOk = LoqActiiveX.Record ("Hello World", "myfile.wav")	Can be either .WAV or RAW file (no header), depending on the file extension. Returns TRUE in case of success
Pitch	Change the pitch baseline	LoqActiveX.Pitch=65	Range: 0 – 100 Default value: 50
Speed	Change the speaking rate.	LoqActiveX.Speed=25	Range: 0 – 100 Default value: 50
Volume	Changes the volume.	LoqActiveX.Volume=40	Range: 0 – 100 Default value: 50
Voice	Changs the speaker voice	LoqActiveX.Stop LoqActiveX.Voice="Susan"	Call Stop method before Voice
Frequency	Change the output audio frequency	LoqActiveX.Frequency=8000	Value: 8000, 16000
Coding	Change the output audio coding	LoqActiveX.Coding="l"	Value: "l", "a" and "u"
GetVersion	Return TTS version string	LoqActiveX.GetVersion	
SetAttribute	Set a TTS Attribute See 13.1 and 13.2 for attribute details	LoqActiveX.SetAttribute Name, Value	Name: a TTS Attribute Name (e.g. "ReadingMode") Value: new attribute value to be set (e.g. "xml")

GetAttribute	Get a TTS Attribute See 13.1 and 13.2 for attribute details	Val = LoqActiveX.GetAttribute Name	Name: a TTS Attribute Name (e.g. “ReadingMode”) Value: a string containing the attribute value (e.g. “xml”)
Enum	Perform a Query to TTS according with “request” and “conditions”	Val = LoqActiveX.Enum (RequestedData, Conditions)	RequestedData: (e.g. “Speaker”) Conditions: comma separated list of conditions (e.g. “Frequency = 16000, Coding = L”) Val: 1st query result
EnumNext	Retrieve Next TTS Query	Val = LoqActiveX.EnumNext	Val: query result data (“” if last)
ReadFile	Read a text file	LoqActiveX.ReadFile(“myfile.txt”)	
GuessLanguage	Guess the language of a piece of text	Val = LoqActiveX.GuessLanguage(“this is a chunk of text in English”)	Return value: a language name (“e.g. “English”, “French”, “German”, “Spanish”, “Italian”, “Greek”, “Portuguese”, ”Dutch”, ”Swedish”, etc.)
GuessFileLanguage	Guess the language of a text file	Val = LoqActiveX.GuessFileLanguage(“myfile.txt”)	same as above
Connect	Connects to a network port. From this moment on, audio will output there instead of the audio board	Val = LoqActiveX.Connect PortNumber	TRUE if successful
Disconnect	Disconnect from a network port	LoqActiveX.Disconnect	

Properties:	**Description**
Coding	Change the output audio coding
Frequency	Change the output audio frequency
Pitch	Chang the pitch baseline (Range: 0 – 100 Default value: 50)
Speed	Change the speaking rate (Range: 0 – 100 Default value: 50)
Volume	Change the volume (Range: 0 – 100 Default value: 50)
Voice	Change the speaker voice
Reset	Reset voice parameters (speed, pitch and volume) to default values
Lexicon	Load a User Lexicon
Language	Language to speak (LoqActiveX.language=”italian”)

Events:	Description
StartOfSpeech	TTS has started
EndOfSPeech	TTS has finished
SpeechPaused	TTS has been paused
SpeechResumed	TTS has been resumed (after a pause)
Bookmark	A bookmark has been encountered in the input text (parameter: bookmark string)
AsynchError	An error has occurred
Sentence	A new sentence has started
Tag	A text embedded tag has been encountered (parameter: tag string)
Phoneme	A phoneme has been produced (parameters: ipacode long, duration in ms - long)
Voice	Voice has been changed via text embedded tag (parameter: voice string)
Language	Language has been changed via text embedded tag (parameter: language string)
Audio	An audio command has been issued via text embedded tag (parameter: command string)

19 Loquendo TTS protection schema

Loquendo TTS uses a license key to protect itself against illegal copy. The license is host-based: each computer has a different license key.

Each license has different capabilities:

- The maximum number of channels (i.e. the max number of instances you can run simultaneously)
- The list of licensed voices
- The list of additional capabilities (e.g. Mixed language Support)
- The expiration date (for evaluation licenses only)

The definition of “maximum number of channels” requires to be explained in more details.

Prior to version 6, Loquendo TTS used to limit the number of channels that could be opened. Now this doesn’t happen anymore. This will be explained in more details in the next section.

19.1 Processing and speaking time

Compared with the speaking time, the TTS processing time is relatively short. This means, for instance, that producing a 20 second audio signal may require a fraction of a second. Due to this a very high number of simultaneous instances can be performed in real-time (even 100 for a new generation processor, such as a 2.4 GHz Pentium IV).

For the same reason, the latency, i.e. the time between a text-to-speech request and the beginning of audio flowing, is very short (a fraction of the first sentence speaking time).

A telephony or multimedia application that outputs the audio directly to the soundcard or board, may take advantage of the Loquendo TTS fast performances, because as soon as TTS starts producing audio data, the board can immediately play it back, without waiting that the entire text has been processed. The latency time, of course, may vary depending on the number of simultaneous instances running, but, even with very high loads and most complex texts, it is guaranteed to be in the order of a second or two.

Loquendo TTS can also save speech samples into WAV audio files to be played back by the application like any other waveform file. This function mode may be preferred in order to speed up the integration (many existing application can manage audio files). However, in this way, the latency time is longer, because the application must wait until the entire audio has been produced before starting playing. Due to Loquendo TTS very high performances, even that amount of latency can be acceptable for most applications, at least in case of not too long sentences.

There is a big difference between the two scenarios above: in the first one (i.e. direct output to board - let’s call it “Real-time mode”), a single Loquendo TTS instance is busy for the whole speaking time – or at least it is busy for *almost* the whole speaking time. See the time diagram below:

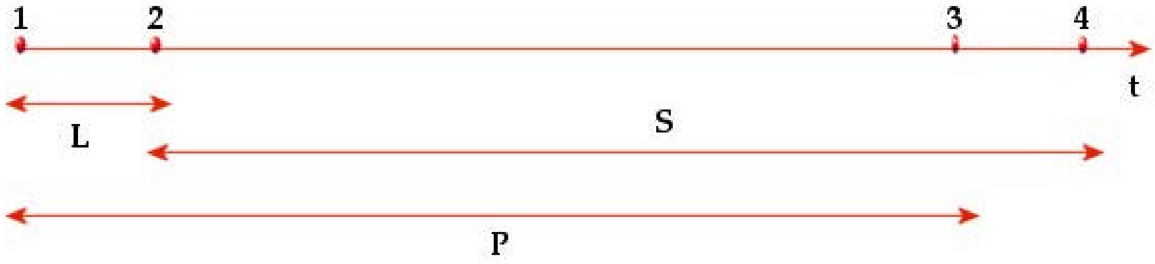


Fig 1: Real-time Mode

TTS conversion is requested at time 1. Loquendo TTS starts processing and the audio starts flowing at time 2. The interval between 1 and 2 is the latency (L). Between 2 and 3 the TTS conversion is

active and, in the meantime, the audio continues flowing toward the audio board.
At time 3 the TTS processing is over, but the audio board will continue playing until time 4 (the 3-4 interval length is the time requested for flushing the audio board internal buffers, and it should be reasonably small, compared with the others). Finally the "end of speech" is reached at time 4 and the audio board becomes silent. The interval 1-3 is the processing time (P), while the interval 2-4 is the speaking time (S).
As you can see from the diagram above, processing and speaking times are nearly equivalent, since the board naturally slows down the TTS process (if the audio data flowed too fast, the audio board internal buffers would overflow). Of course, there is a very little CPU consumption because its occupation is distributed during the processing time (P). This makes the Real-time mode the recommended one for complex multi-channel applications. In addition, the TTS conversion is really an interactive process: the application can stop, pause, and resume speaking, skip forward or backward, receive synchronization events from the TTS, for the whole duration of the TTS process. We may say that what you hear is what it is currently being synthesized.
Let's consider now the second scenario (let's call it "Batch mode"), with output to a waveform file.
See the diagram below:

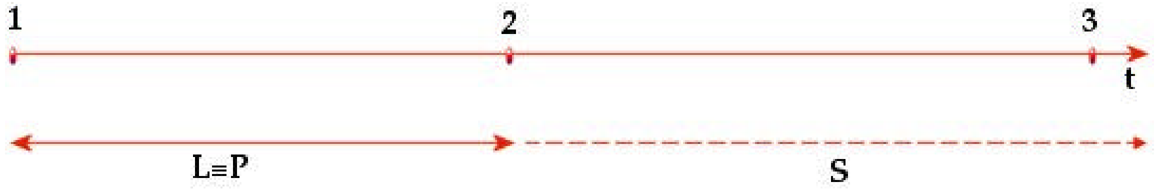


Fig 2: Batch Mode

Supposing that the TTS conversion starts at time 1, at time 2 the WAV file ha been completely written. The processing time (P) and the latency (L) coincide (interval between 1 and 2) because the application must wait for the entire file has been written before starting playing. Since the audio playing is performed by the application and not by the TTS, the latter has no control over it. For this reason, the TTS cannot stop the audio playing and there is no chance of synchronizing text and audio, etc. (and for the same reason the speaking time line S is dashed in the diagram above).
During processing time, there is a high CPU consumption, because the TTS conversion runs at the maximum speed, so the batch mode is not recommended for multi-channel applications (strong CPU peeks can affect the global performance).
However, the processing time is but a fraction of the speaking time. This means that the TTS can keep on processing several new utterances while the system is still busy reading the very first one.
Consider now that every Loquendo TTS license has a limitation in its maximum number of channels (that is the maximum number of simultaneous instances you can run) and, of course, a 30 channels license costs more than a 4 channels one. But what does "running instance" mean?
Let's suppose for a moment that an instance may be considered "running" for the whole duration of its processing time. A "batch mode" application could reuse the same instance for producing more than a file, will the system is still busy reading the first one : remember that the ratio P/S could be 1/50 or even less!
We usually call that technique "channel multiplexing": as soon as message no. 1 is ready (this happens a time 2 in the diagram above), the instance is immediately available for processing message no. 2: in the meantime the application can start playing message no. 1. Before the end of play, many other messages could be produced.
Instead, an instance has to be considered "running" for the whole speaking time, even if the processing time is considerably shorter (of course, there is no ambiguity with Real-time mode, as in that case processing and speaking times are equivalent).
Does it means that TTS should be slowed down to avoid channel multiplexing? No, it would be clearly unacceptable. Let's describe how this has been implemented.

19.2 Instances and "tokens"

Suppose that we own a four-channel Loquendo TTS license. We know that we can run no more than 4

simultaneous TTS instances. However we can open as many TTS instances we like, provided that no more than four of them are running at the same time.
The system reserves a pool of 4 “passes” (we call them “tokens”) than can be assigned to running instances. As soon as a token has been assigned, no more instances can borrow it. However each token has its own expiration time, after that time the token will be again available.

As soon as instance no. 1 starts running (let’s say at time t_1), the first available token (e.g. token no. 1) is assigned to that instance. Token no. 1 will be available again at time t_1+s_1 (where s_1 is the current speaking time for instance no 1, point 3 in Fig 2) even if the processing ends very earlier (t_1+p_1) (this can happen only in batch mode, since in the Real-time mode there is no real delay between the end of processing and the token expiration time). Instance no. 1 (as well as any other instance) can immediately borrow another token (e.g. token no. 2) that will expire at time (t_2+s_2). If all tokens have been already assigned, no more TTS instances can run. Any pending TTS request will be delayed until a new token will become available.
Note that the tokens can be shared among all TTS processes running on the system.
As far as a token is available, any batch mode TTS runs always at its maximum speed. However with a 4-channel license you cannot overcome the *average* speed of 4X.

19.3 What is the best mode: Real-time or Batch?

It all depends on the system and application requirements. The Real-time mode is mostly suitable for real time applications where the audio produced by the TTS engine is streamed into a soundcard or a board. In this case the TTS synthesis is a synchronized process that make a little use of the CPU in an uniform manner during the entire process. The batch mode case is more suitable for concentrated (channel multiplexing) sort of applications, where the CPU is stressed for a shorter time (point 2 in Fig 2) and when interaction is not required between the synthesis process and the application

Nevertheless whenever mode of operation is chosen the CPU processing time is nearly the same.

20 APPENDIX A: Software redistribution

In order to ship Loquendo TTS in conjunction with a custom application, further steps are necessary. This is a list of files the installation procedure should manage:

Copy and execute on the target machine file hldrv32.zip (if you want to add the hardware plug support) – Win32 only.

> This is the hardware plug installation program. Administrator rights are required in Windows in order to run this application

Copy the following files to the target directory **using the same directory structure as in your Loquendo SDK installation**:

1. All files with prefix "Loq", whose name is a language (e.g. LoqEnglish6.0.dll) and whose extension is .dll (Windows) or .so (Linux)
2. All files with extension ".lex" (e.g. EnglishUs.lex)
3. All files whith extension ".atm" (e.g. EnglishUs.atm)
4. All files with extension ".phd" (e.g. EnglishUs.phd)
5. All files with extension ".bin". To save space you can copy only *SpeakerRateCodingProc*.bin (replace the italics according to the table below)
6. All files with extension ".sde".
7. All files with extension ".pmk".
8. All files with extension "*Language*.lde" and "*Speaker*.vde".

Language	SpanishEs, EnglishUs, EnglishGb, German, PortugueseBr, Greek, French, Italian, Swedish, Catalan, SpanishAr, etc.
Speaker	Roberto, Marcello, Mario, Gabriela, Silvana, Valentina, Elizabeth, Juan, Bernard, Sophie, Susan, Kenneth, Ulrike, Artemis, Carmen, Diego, Esperanza, Francisca, Annika, Montserrat, Paola, Luca, Simon, LinLin, etc.
Rate (sample rate)	08 for 8 kHz; 11 for 11 kHz; 16 for 16 kHz;
Coding	L for Linear PCM, a for A Law, u – for µ Law
Proc	i for Intel™ m for Motorola™

For instance *Roberto16li.bin* means Loquendo™ TTS male Italian voice 16 kHz linear PCM for Intel™.

20.1 Loquendo™ TTS Gilded modules

Some of the most recent distributions contain the "Gilded" extension, which is a collection of Expressive Cues.

The repertoire of *Expressive Cues* consists of a set of pre-recorded formulas, comprising conventional figures of speech, like greetings and exclamations ("hello!", "oh no!", 'I'm sorry!"), interjections ("Oh!", "Well!", "Hum"..) and paralinguistic events (e.g. breath, cough, laughter, etc.), which suggest expressive intention (to confirm, doubt, exclaim, thank, etc.).

In order to ship the “Gilded” modules in conjunction with the standard voice modules, further steps are necessary. Copy the following files to the target directory using the same directory structure as in your Loquendo SDK installation:

1. All files with extension “.gde”
2. All files with extension “.eps”
3. All files with extension “.epd”
4. All files with template “*Gilded*.bin”. To save space you can copy only *SpeakerGildedRateCodingProc*.bin (replace the italics according to the previous table)
5. All files with template “*Gilded*.sde”.
6. All files with template “*Gilded*.pmk”.

20.2 Loquendo™ TTS DLL

Copy the following files to the target directory:

LoqTTS6.dll (Windows) or LoqTTS6.so (Linux)
LoqLanguageGuesser6.dll (Windows) or LoqLanguageGuesser6.so (Linux) (optional)
LoqTTS6_util.dll or LoqTTS6_util.so (Linux)
LoqAudioBoard.dll or LoqAudioBoard.so (Linux) (for audio board playing)
LoqAudioFile.dll or LoqAudioFile.so (Linux) (for PCM file production)

Register LoqTTS6.dll[16]:
Run *regsvr32.exe LoqTTS6.dll*

20.3 Loquendo™ TTS ActiveX[17]

Copy and register also LoqActiveXW.ocx, by running: “*regsvr32.exe LoqActiveXW.ocx”*

20.4 Loquendo™ TTS SAPI 5 support[18]

Copy and register also LoqSapi5.dll, by running: “*regsvr32.exe LoqSapi5.dll”*

This will register all the Loquendo TTS voices to be used with SAPI5. Note that the SAPI5 redistribution package (see Microsoft SAPI 5 documentation) may also be included

[16] Windows only – this will create the [default.session] section and the write the correct DataPath value
[17] Windows only
[18] Windows only

20.5 Loquendo™ TTS SAPI 4 support[19]

Copy LoqSapi4.dll and LoqAudioSapi4.dll in the target directory. Register LoqSapi4.dll, by running: *"regsvr32.exe LoqSapi4.dll"*

This will register all the Loquendo TTS voices to be used with SAPI4. Note that the SAPI4 redistribution package (see Microsoft SAPI 4 documentation) may also be included

[19] Windows only

21 APPENDIX B: FAQ and Troubleshooting

- **You've just installed Loquendo TTS: Edit2Speech reports that the license code is missing**

Run the program TTSLicense from the Start Menu, press Help button and follow the instructions

- **Edit2Speech reports that no voices are installed**

Open the registry editor (REGEDIT) and check the following branch:
[HKEY_LOCAL_MACHINE\SOFTWARE\Loquendo\TTS\Edit2Speech]
You should see a keyword similar to:
"DataPath" = "c:\Program Files\loquendo\LoquendoTTS"[20]
Be sure that the correct path to Loquendo TTS files is specified.

Run the TTSDirUpdate program included with the SDK
If this does not fix the problem, reinstall the program

- **You have installed two versions of Loquendo TTS on the same PC with two different licenses and it doesn't work**

At the moment, a single computer cannot host more than one Loquendo TTS instance. A single computer cannot host more than one Loquendo TTS license. These limitations may be removed in future releases

- **You have renamed or moved the Loquendo TTS folder and now it doesn't work anymore**

If Loquendo TTS folders have been moved or renamed the TTS engine will be unable to resolve the links to its data files and therefore it won't work anymore. Sometimes this is the consequence of multiple install / uninstall / reinstall actions.In this case the best thing to do is uninstall everything and manually delete all references to Loquendo TTS, before reinstalling again.

To do so, uninstall Loquendo TTS the usual way. Eventually delete all residual files from Loquendo TTS directory: (usually C:\Program Files\Loquendo\LoquendoTTS) and then manually remove that folder. Eventually delete all residual files from Loquendo SAPI5 support directory: (usually C:\Program Files\Loquendo\LoqSAPI5) and then manually remove that folder.

Open the Windows registry editor. From the [HKEY_LOCAL_MACHINE\SOFTWARE\Loquendo\TTS] branch, remove anything but the "LicenseCode" key (if you remove "ActorLicense" you will need to install your license again). From the [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Speech\Voices\Tokens] branch, remove all keys with a "LQ" as prefix (e.g. LQSusan).

Now you are ready to reinstall Loquendo TTS.

- **You do not hear sound, even if Loquendo TTS seems to be correctly installed**

Most of the samples included with Loquendo TTS require a Sound Blaster or any other card compliant to Windows Multimedia Standard. If you aren't sure that such a card exists in your system, open the Windows Media Player (or other sound program that you may have installed) and check if you can run it successfully. Note that some old cards are unable to play sound formats other than linear PCM. If you cannot determine your sound card capabilities, try to install one or more Loquendo TTS linear 16 khz PCM voices.

[20] depending on the path you've installed Loquendo TTS to.

- **You want to develop your own audio destination, but you don't know where to start from**

Read carefully section 17.1 and start working from the "LoqAudioFile" audio destination source code.

- **The voice output sounds noisy or corrupted**

Check if you are using the correct sample format (linear PCM, G711 A-law/µ-law) and possibly try to change it. Be sure the audio coding you specify in the ttsNewVoice call matches the one specified in the ttsSetAudio call.

- **Loquendo TTS complains about the lack of a file, but you don't know what file is missing.**

Try to enable logging. Open the registry editor and insert the following keyword:
"LogFile" = "MessageBox"[21]
under section [HKEY_LOCAL_MACHINE\SOFTWARE\Loquendo\TTS\default.session]
You may want to redirect logging to a file instead, specifying a valid filename, e.g.:
"LogFile" = "c:\mydir\myfilename.log"
For a more detailed logging you can also use the keyword:
"TraceFile" = "c:\ mydir\myfilename.log"
Do not use MessageBox in conjunction with TraceFile, otherwise you'll receive dozens of message boxes.
Note that you must restart your Loquendo TTS session, in order to make your changes effective.
Remember also that Loquendo TTS searches all its files in the directory specified by the DataPath keyword in the registry or INI file
This keyword has been setup by the Loquendo TTS installation procedure, and has normally the following value:
"DataPath" = "c:\program files\loquendo\LTTS"

- **(Unix only) You have loaded the LoqTTS shared object with the function dlopen, and the system give you the message "Unresolved external symbol ..." .**

You must add RTLD_GLOBAL to the mode parameter of dlopen: see dlopen man page for details.

[21] This works in Windows only. Try "stderr" instead, if you are working in other environments. The INI file behaves exactly the same way as the Windows registry.